Performance Analysis of Multiprocessor Real-Time Systems with Shared Resources

Performanzanalyse von Multiprozessor-Echtzeitsystemen mit gemeinsamen Ressourcen

Von der Fakultät für Elektrotechnik, Informationstechnik und Physik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung der Würde

eines Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

Von: Simon Schliecker aus Aachen
Eingereicht am: 10. August 2010
Mündliche Prüfung am: 27. Januar 2011

Referenten:
Prof. Dr.-Ing. Rolf Ernst
Prof. dr. ir. Marco Bekooij
Prof. Dr.-Ing. Harald Michalik (Vorsitzender)

2011

Abstract

Consumers increasingly value the benefits of cyber-physical systems to make their life safer, more productive, and also more comfortable. In order to service the growing amount of complex, often distributed functions, many embedded systems today for example in automotive are actually networked systems that consist of multiple independently scheduled processing nodes. Additionally, the computing workload imposed by todays and future applications implies that the nodes themselves are already becoming complex systems consisting of multiple processors and shared resources. This hierarchical setup leads to a significant system complexity.

To maintain the productivity during the development process and to ensure the safety during the product's deployment, additional measures have to be taken that go beyond the classical approach of programming, simulation, and debugging. Through its abstract nature, formal analysis concepts are an ideal tool to identify integration problems early in the development process, and through its proven correctness defuse the risk of component misdimensioning that would endanger the safety of the system and its user at run-time. In this thesis, we take on the integration challenges introduced by todays and upcoming architectures by providing a suitable performance analysis that allows accurate predictions in different phases of the design process.

For this, the focus will first be on the timing properties of networked multi-node systems for which we revisit established performance metrics known from previous research. New methods are introduced to conservatively determine the distorted patterns of events along a processing chain, the tasks' worst case response times, and the latency via a chain of tasks in setups where the timing constraints are specified end-to-end. These metrics are derived on the basis of a new abstraction to represent the performance of a single task that very efficiently captures its relevant timing properties. The new model is a straight-forward extension based on classical scheduling analysis using busy windows, and can thus be provided for a large spectrum of established scheduling policies. Together with a generalized treatment of event models to consider also complex event patterns, the proposed methods lead to very accurate results in a large range of setups, notably without sacrificing efficiency.

The growing processing and safety requirements lead to the introduction of multi-core components, increasingly also in control-dominated systems. The integration and verification of multiple tasks on such an architecture is a challenge in itself, aggravating the system-level integration challenges: The timing of tasks here is highly interdependent — even if they are mapped to different cores. Resources such as mem-

ories, coprocessors and system-bus interfaces are mostly provided per-chip and will be shared by all tasks. This causes a run-time competition that has to be arbitrated both fair and efficiently. But the most efficient arbitration policies rely on dynamic run-time decisions, which makes a prediction of the inter-core timing interference challenging.

To tackle such setups, a methodology is established in this thesis that breaks down the complexity through the introduction and investigation of new performance metrics and corresponding analyses: First, the run-time load of each task and processor on the shared resource is formally investigated. The results of this analysis then allow the derivation of the resulting latency of the shared resource operations. Finally, the classical task scheduling analysis is extended to accommodate these delays. This decomposition makes the derivation of task response times in a multicore setup feasible, and has the additional advantage of separating the concerns of scheduling and resource arbitration.

In order to facilitate this analysis in the presence of analysis dependencies, a compositional analysis approach is adopted and generalized. This thesis drops the predominant focus of previous approaches on task-activating event models, and lays out the means to integrate any combination of interdependent analysis functions for various, heterogeneous system parameters, such as the ones introduced by multicore components. Together with the refined analyses of existing metrics and novel analysis components for multicore this framework represents a versatile tool for the performance analysis of heterogeneous multiprocessor systems.

Kurzfassung

Eingebettete Systeme, die das Leben sicherer, produktiver und komfortabler machen, werden von Konsumenten in immer stärkerem Maße wertgeschätzt. Viele dieser eng mit der Außenwelt interagierenden Systeme zum Beispiel im Automobil bestehen schon heute aus zahlreichen Einzelkomponenten, die gemeinsam die komplexen, oft verteilten Funktionen bereitstellen. Um den wachsenden Rechenanforderungen gerecht zu werden und möglichst auch die Anzahl der Recheneinheiten zu reduzieren, sind die einzelnen Komponenten inzwischen selbst komplexe Systeme, bestehend aus mehreren Prozessoren und gemeinsam genutzten Ressourcen. Dieser hierarchische Aufbau führt zu einer nur schwer zu beherrschenden Systemkomplexität.

Um dennoch die Produktivität im Entwicklungsprozess und die Sicherheit der ausgelieferten Systeme zu gewährleisten, werden Methoden benötigt, die über die klassischen Verfahren wie Entwicklung, Simulation und anschließender Fehlersuche hinausgehen. Formale Analysemethoden sind eine ideale Ergänzung um die aufkommenden Integrationsrisiken frühzeitig zu erkennen und zu entschärfen, und um insbesondere in sicherheitskritischen Systemen eine Unterdimensionierung von Systemkomponenten zu vermeiden, die zu einer Verletzung von zeitlichen Anforderungen führen würde. In dieser Arbeit begegnen wir den Herausforderungen bestehender und zukünftiger Systemarchitekturen mit einer formalen Methode zur Performanzanalyse.

Dabei liegt der Fokus zunächst auf der Analyse der verteilten Funktionen, indem auch etablierte Performanz-Indikatoren erneut untersucht werden. Es werden neue Methoden eingeführt, um das zeitlich verzerrte Verhalten von Ereignissen entlang einer Task-Kette zu untersuchen und um die Antwortzeiten der Tasks und die Gesamt-Latenz entlang einer Kette zu bestimmen. Diese Methoden basieren auf einem neuen Modell des Zeitverhaltens eines einzelnen Tasks, das sehr effizient die nötigen zeitlichen Eigenschaften abstrahiert und diese für die System-Analyse zur Verfügung stellt. Zusammen mit einer verallgemeinerten Betrachtung von Event-Modellen, die auch spezielle Charakteristika wie periodische Bursts erfasst, führen diese Methoden zu sehr präzisen Ergebnissen ohne Effizienz einzubüßen.

Die wachsenden Berechnungs- und Sicherheitsanforderungen führen auch in Steuerdominierten Systemen vermehrt zum Einsatz von Multicore-Controllern. Die Integration und Verifikation von Tasks auf einer solchen Plattform stellt wiederum ein Problem dar, das den System-Integrationsprozess um eine Komplexitätsstufe anreichert: Das Zeitverhalten der Task ist voneinander abhängig, auch wenn sie unterschiedlichen Prozessorkernen zugewiesen werden. Ressourcen wie Speicher, Coprozes-

soren, oder Bus-Schnittstellen stehen pro Chip zur Verfügung und werden von allen Tasks gemeinsam genutzt. Dies führt zu Laufzeit-Konflikten, die fair und möglichst effizient aufgelöst werden müssen. Um eine hinreichende Auslastung zu erzielen, werden Zuweisungsentscheidungen dynamisch zu Laufzeit getroffen. Dies jedoch macht eine Vorhersage des möglichen Zeitverhaltens äußerst schwierig.

Um solche Systeme trotzdem in sicherheitskritischen Umgebungen einsetzen zu können, sind neue Methoden nötig. Der Lösungsvorschlag dieser Arbeit ist, die Problemkomplexität durch Aufbrechen in mehrere dedizierte Analyseschritte beherrschbar zu machen: Zunächst wird die Last bestimmt, die ein Task oder ein Prozessor zur Laufzeit auf eine gemeinsam genutzte Ressource verursachen kann. Basierend darauf kann dann die Latenz der Operationen unter Berücksichtigung des Arbitrierungsverfahrens bestimmt werden. Schließlich fließen diese Latenzen in eine erweiterte Antwortzeitanalyse ein. Diese Aufteilung ermöglicht die Analyse solcher Multiprozessorsysteme und hat noch einen weiteren Vorteil: Unterschiedliche Scheduling- und Arbiterierungsverfahren können kombiniert werden, so dass diese und die zugehörigen Analysemodule einzeln optimiert und verfeinert werden können.

Um die Analyse der verschiedenen Komponenten auch bei Vorhandensein zyklischer Abhängigkeiten zu ermöglichen, wird in dieser Dissertation ein kompositionaler Ansatz gewählt. Vorhergehende Arbeiten, die sich weitestgehend auf die zeitlichen Eigenschaften Task-aktivierender Ereignisse konzentrierten, werden aufgegriffen und es wird dargelegt, wie eine Menge heterogener Modelparameter und zugehöriger Analysefunktionen miteinander kombiniert werden kann. Zusammengenommen ergibt sich durch die verbesserte Analyse bestehender Metriken und die Bereitstellung neuer Module für Multicore ein flexibles Analyseframework für heutige und zukünftige Multiprozessorsysteme.

Acknowledgments

This thesis would not have been possible without the extensive support and valuable input from my colleagues, friends, and family.

My sincere gratitude goes to Professor Rolf Ernst, for securing my path through the challenges of the academic world. I have benefited in numerous ways from his deep technical and personal advise and his invaluable ability to bring together great people.

Many of the great minds working around my area of research, I had the opportunity to meet, and in some cases I was especially privileged to work together with. In particular, I have considered the industrial projects with Pierre Paulin from STMicrolectronics, and Professor Gabriela Nicolescu from the École Polytechnique de Montréal, Paolo Giusto from General Motors to be a valuable complement to my academic research, and would like to thank them for their support and input. I would like to thank Kai Richter and Marek Jersak for their personal advice and the insight into the growing importance of formal methods in industrial practice as demonstrated through the Symtavision venture.

A special experience for me was my stay at Philips Research in Eindhoven (now in part NXP Semiconductors), from which I greatly benefited through the years, and most likely years to come. In particular, I am indebted to Professor Marco Bekooij, who has enriched my academic view through advice and discussions, and who now kindly agreed to become part of my doctoral committee. I am also thankful for the great time in joint research projects and the valuable input from fellow researchers at the ETH Zürich, TU Eindhoven, and Linköping University.

Through the excellent collective of colleagues at the Institute for Computer and Communication Engineering in Braunschweig, it has served as a fabulous basis to enable this work. I am therefore indebted to the great technical and administrative team (including Tina Boettger, Sabine Klöpper, and Ina Niedermayer). My gratitude goes to all colleagues for providing continuous support, input, and fun time. In particular, I would like to thank my fellow researchers Steffen Stein, Mircea Negrean (thank you for the many excellent discussions and productive sessions), Matthias Ivers, Jan Staschulat, Jonas Rox, Arne Hamann, Razvan Racu, and Rafik Henia, as well as Judita Kruse, Jörn-Christian Braam, Amilcar Lucas, Sven Heithecker, Ruben Jubeh, Moritz Neukirchner, Daniel Thiele, Jonas Diemer, Henning Sahlbach, Matthias Hanke, Sean Whitty, Philip Axer, Maurice Sebastian, Christoph Ficek, Harald Schrom, and Tobias Michaels — all of whom I also wish the best for their academic and industrial endeavours.

I was also very lucky with the students that have deemed my research interesting enough to make it their own. The present thesis has particularly benefited from the the input of Octavian Paunescu, Giuliano Lima, and Axel von Engel, and some that were already named above that to my great joy have now become friends and colleagues.

I would like to thank my parents Ingrid and Gerd. Without their love and support through my studies this academic endeavor would not have been possible. Finally, I can not sufficiently express my gratitude to my wife Verena. Thank you for keeping me sane and happy.

Contents

1 Introduction

1.1 Motivation

Across a broad range of industries and markets, an increasing share of a product's value is provided by embedded computer systems. This trend is supported by the continued innovation in industry and academia that envisions new electronical functions on top of what can be achieved by electrical or mechanical measures alone, and that provide more cost-efficient implementations of formerly physical designs. The new functionality is eagerly awaited on the markets by a steady consumer demand for lower cost, more reliable, and more feature-rich products.

The added electronical features inevitably require more computational resources that allow performing more complex algorithms and generally run more sophisticated applications in a real-time environment. In the automotive industry this approach has led to the introduction of numerous embedded control units (ECUs) that each contribute a specific functionality to the car. As long as the number of such functions was small, and the functions were mainly tied to the mechanical counterparts which they control, this procedure was feasible. But by the mid-2000s it has led to more than 70 distinct controllers in a single high-end vehicle that must communicate over a set of field-buses and dedicated interconnects [Bro07]. The integration of some of these functions on a reduced set of controllers promises a tremendous cost saving potential.

The requirements for future computing platforms are thus to enable the integration of multiple functions onto a single ECU, to increase the computing performance provided to each application, and to deliver higher system reliability. These goals can not be achieved with the traditional, single-processor ECU design, because it does not scale without entailing new challenges: For example, increasing the processor's frequency leads to a quadratic increase in the component's power consumption [Cha92] which generates even more concerns about the required energy and heat dissipation, and raises issues with regard to electro-magnetic compatibility.

Consequently, the industries are turning to multicore solutions to deliver the necessary performance. Multicore designs have been used successfully in data-dominated systems, such as multimedia, or around applications that are by nature parallelizable, such as in server-based data centers. According to [Gul07], the shipping volume of multicore processors was expected to quadruple in the time from 2007 to 2009 and continue to do so through 2011. Although this development was offset by the global

recession, recent studies indicate that it is now back on track [Sch09a]. The application of multicore processors in control-dominated systems has focused on heterogeneous setups [Fre00, Inf08], in which a dedicated architecture can suitably be used to address a mix of low-latency and high-bandwidth service requirements. Recently, also homogeneous multicores have been proposed [Fre09a, inf] for use as powerful computing platforms.

In order to exploit the technological advantages of such new technologies for real-time systems, it is mandatory to have a thorough understanding of the resulting timing. Any failure to accurately predict the behavior of the final product will lead misdimensioning. In the best case this causes annoying cost increases, but in the worst case it endangers the correct operation and introduces safety risks.

The integration of multicore components into distributed cyber-physical systems leads to highly heterogeneous setups that consist of several processing nodes that are connected via a communication network. To provide sufficient computing performance, the nodes themselves are complex systems, sometimes consisting of multiple processors with a local bus and shared resources. The resulting hierarchical setup entails a multitude of challenges with respect to the timing predictably and indirectly the established design process. In this work, we provide answers to the question on how the relevant performance guarantees can be established in such setups.

This thesis introduces a formal performance analysis for today's and future networked embedded systems with real-time requirements. The analysis can accompany the development process during system design, component dimensioning, performance verification, and product certification. The performance predictions are not only safe, but also accurate by addressing the versatile behavior of typical embedded systems.

In the remainder of this chapter, we will take a closer look at the predominant trends in the embedded systems industry in order to project the hardware and software architectures of today's and upcoming designs (Section 1.2). We then highlight the classical steps in the development process in Section 1.3, which allows us to see how timing problems are classically considered. The challenges to this process that are introduced by the new architectures are then identified in Section 1.4.1. This leads us to a discussion of the previously proposed countermeasures in Section 1.5 and the identification of the benefits of our methodology in Section 1.6.

1.2 Embedded System Trends

In the past years, the value generated in embedded systems has grown in many, if not all economically important domains [ZVE09] such as industrial automation, consumer electronics, or medical technology.

In the following section, we take a closer look at the situation in the automotive industry. This industry is subject to representative trends that can also be observed in other domains: The strict constraints of embedded systems are combined with the

large markets of consumer electronics and a general openness to evolving technologies in light of a global competition.

1.2.1 Market Trends

A growing share of the value generated in a modern car is created in its electric, electronic, and programmable electronic (E/E/PE) components. The value of electrical systems and electronics in the average automobile will in 2015 approach EUR 4,150 (as opposed to EUR 2,220 in 2004) [Dan04]. This trend is reflected in the continued growth of the automotive electronics market that continuously beats the growth of the overall vehicle markets as shown in Figure 1.1. The growth could be sustained even during the recession of 2008/2009, and is expected to continue to average around 7% annually [Sch09a].

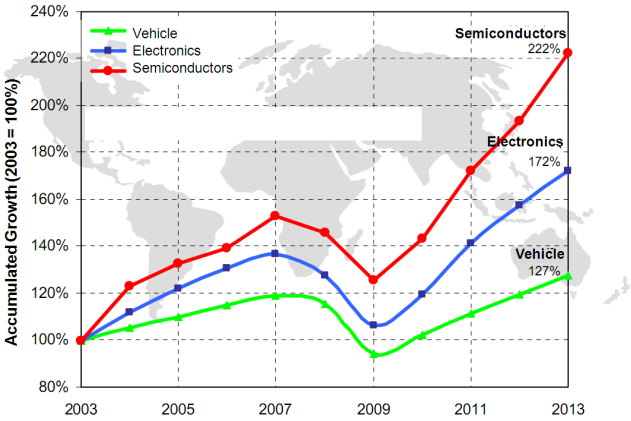

Figure 1.1: Relative Growth of Semiconductors, Embedded Modules and Vehicles Markets in Automotive (Source [Sch09a])

Several major trends can be identified that indicate that the amount of electronic and software-based solutions will significantly grow in the future. To accommodate the additional software, powerful controllers are required that deliver the computing power, without sacrificing non-functional requirements such as power, reliability, or electro-magnetic radiation.

Reducing the Environmental Impact Consumers and lawmakers worldwide are becoming increasingly aware of the scarcity of common natural resources and the environmental impact of its uncontrolled exploitation. The transportation sector is

a significant contributor to the release of recognized climate gases[1]. In Germany 2007, one third of the country's energy was consumed by transportation, with more than 80% of it attributed to road traffic [Sta09b]. This has led to stringent emission regulation in this area [Rod10].

A key concept to tackle this challenge is the optimization of the combustion engine's control to allow a more fuel-efficient injection, and further innovations in the powertrain platform (start-stop technology [Wei07], regenerative breaking, and the move towards hybrid or even fully electrical engines). These developments imply more complex control applications to regulate the physical components. Another effective method to cut the fuel consumption is to reduce the vehicles weight [Bun07]. This can be achieved by replacing mechanical and hydraulic control schemes by electronic X-by-wire solutions [Lee02] — which again increases the control complexity. Weight is also saved by decreasing the number of ECUs and in the car, which to a certain extend allows streamlining the network topology [Obe09]. All of these means directly or indirectly demand for the availability of more computing power per control unit.

Increasing Passenger and Pedestrian Safety Mechanical measures to increase the passengers safety have long been explored and are mostly exploited [Lee02]. Additional improvements can only be achieved with active measures that better assist the driver in avoiding collisions in the first place. The complexity of such measures reaches from simple break control [Lei80] to camera-based lane, pedestrian, and object recognition systems. The recent and upcoming applications consist of several sensors and actors, and significant data processing in between [Cur00]. The introduction of high data volumes into the domain of reliable control applications makes this field particularly challenging. Not least does it lead to an increased stress on the underlying computing hardware and design methodology.

Increasing the Car's Reliability The electrical and electronical components are responsible for a rising share of car breakdowns. The German automobile association (ADAC) identified the electronics as the source for car breakdowns in 40% of the cases in 2007[2] (up from 35% in 2003[3]). This highlights that the correct vehicle operation increasingly relies on the correct functioning of the applied electronics hardware and software.

One has to be aware that the required microelectronics progress with respect to performance, low power, and cost relies on the increase on the amount of available transistors and a reduction of the feature size. These improvements are lately bought at the cost of a growing susceptibility to transient faults that may directly lead to erroneous internal states [Mat97, Shi02, Bor05]. To compensate for this tendency, the

[1] `http://www.ipcc.ch/` (retrieved 2010-05-23)

[2] `http://www.sueddeutsche.de/auto/adac-pannenstatistik-autos-die-aerger-machen-1.411175-2` (retrieved 2010-05-23)

[3] `http://www.sueddeutsche.de/auto/adac-test-pannenursache-nr-die-elektrik-1.569566` (retrieved 2010-05-23)

computation and communications have to be secured with additional error-detection and correction layers. This will again lead to an increased communication and computation requirement [Man07, Seb09]. To accommodate the additional computation, multicore solutions have been proposed [Smo06, Wel09]. These solutions have the added benefit that they introduce a new dimension of spacial redundancy by separating the operations not only over time, but also over physical cores, which reduces the susceptibility to permanent faults. Besides the said positive economic, environmental and reliability benefit, the envisioned reduction of ECUs also leads to a reduction of other error sources – for example, in automotive environments more than 30% of electrical failures are ascribed to connector problems [Pet06] (citing [Swi00]).

Keeping the Cars Affordable With all these upcoming challenges in mind, car's have to remain affordable. If anything, the global recession of 2009 has increased the attention given to the the cost-efficiency of the automobile products.[4] But changes are also happening on a global scale [May08]. A new segment, the "ultra low-cost vehicles" with price points of less than USD5000 per vehicle is emerging.

As a large share of the value is created in electronics (see above), this segment also has to bear a significant share of the cost pressure. Fortunately, the current E/E/PE topologies provide a vast amount of optimization opportunities. A modern car can easily carry around more than 60 ECUs that are interconnected via a set of 6 field buses [Gri03]. Significant cost saving can be expected if the car's (software) functionality can be implemented on a reduced number of ECUs. In [Obe09] it was demonstrated how quickly an increased component cost is offset by the reduced number of components and the decrease in the wiring effort. Software standardization initiatives such as AUTOSAR [aut06] make this consolidation feasible.

Delivering Improved Customer Experience Finally, customers are to a growing extent perceiving the value of sophisticated electronic and software functions. Then-DaimlerChrysler estimated in 2003 that 80% of all future automotive innovations will be driven by electronics, 90% of which attributed to software [Gri03]. This trend is also indicated by the strong efforts by the OEMs to use built-in navigation and infotainment systems for brand differentiation [Sch09a].

But the standards and applications in these areas are subject to much shorter innovation cycles than given by the average car's lifecycle. Updates are now commonly applied during a product cycle, and also in the field. This strongly suggests software-based solutions on versatile platforms even in domains where dedicated processors and architectures have been predominant (as in [Moo05]).

The cited trends will cause a significant increase of the software and communication load in the automotive electronics that can not be handled by the past approach, in

[4] This has in particular hit the high-class segment, which classically acts as a door-opener to new technologies [Sch09a]

which the number of control units in a vehicle grows proportionally with the number of functions. The more complex applications and higher functional integration demands for sophisticated platforms that deliver the necessary computing power — without sacrificing the non-functional constraints of power, reliability, and weight.

1.2.2 Technology Trends

Finding an optimal architecture is a challenging task due to the stringent constraints of embedded, often mobile, systems that precludes many design options that are available to general purpose computing. A typical embedded design specification demands for functional correctness and correct timing, sufficient reliability, adherence to power constraints, and robustness to future design changes. This diversity of requirements often leads to heterogeneous solutions that are tailored to the application.

Dedicated Hardware Solutions The challenge of a large computing workload can for example be tackled with dedicated hardware, in particular when the required operations are highly regular and sufficiently parallel. The spectrum of implementations reaches from FPGA (field-programmable gate array) based solutions to the manufacturing of dedicated ASICs (application specific integrated circuits), which are not reconfigurable. FPGA's have been proposed for several niche aspects in the automotive electronics infrastructure, e.g. in gateways [San07] and image processing [Ang08, Won10]. The online reconfigurability of such systems is an increasingly interesting aspect, because many applications traverse through a number of dedicated scenarios during a typical drive (e.g. the image processing may be different in the modes "parking", "cruising on highway", or "cruising in rain").

Compared to FPGAs, dedicated ASICs promise a lower power consumption and faster processing speed at a lower cost per unit. However, the setup cost is relatively high, so that a large amount of identical units must be expected. Consequently, this approach is chosen typically only for low-level operations (such as bus controllers, sensor interfaces, ...). In particular, the applications projected in Section 1.2.1 commonly exhibit a complex and irregular control flow, making the dedicated hardware solutions inappropriate.

Programmable Platforms But there are also other design aspects to consider apart from the provisioning of pure processing power. Many high-level automotive functions have been implemented in software (e.g. the engine control, gateway functions, ...), although in theory other approaches would have been possible. This leaves the industry with a large base of existing solutions and significant know-how in the domain to which future solutions must be compatible.

However, continuously increasing the processor's clock speed is not feasible, because this usually implies also adopting the supply voltage to higher levels, which leads to an unreasonable surge in the overall power consumption [Cha92], which quickly introduces thermal problems [Fen03]. Moreover, the higher clock frequency yields

more electromagnetic radiation and susceptibility [Not10]. Thus on the one hand, power and clock constraints are increasingly strict [Sch97], but on the other hand the number of transistors continues to grow [Moo65, ITR]. This suggests measures to "translate transistors into performance" [Fly05]. The driving logic behind this trend can be observed in just about every computing domain: the desktop environment [Gee05], in server and super-computer environments, in embedded consumer devices, and in mobile devices.

One approach is to invest transistors to make the processor pipeline faster (i.e. longer) and better utilized by implementing measures such as speculative execution and prefetching that are often beneficial for the average-case throughput. Besides the debatable benefit of average-case improvements for real-time systems, this path has already been explored to an extent that makes any incremental improvements tremendously complex to design, program, or verify. A more predictable approach is to augment the processor pipeline with a configurable hardware unit. This approach is followed in the application specific instruction-set processors (ASIPs) [Raz94], which attempt to efficiently combine software execution with (reconfigurable) hardware. The approach allows to speed up and reduce the power requirement of the most common functions and instructions. Alternatively, the additional chip area can also be invested into fast memories (e.g. caches) in order to tackle the problem of the increasing speed gap between processing performance and memory bandwidth [Wul95].

Finally, parallel processing has been identified as the key option to tackle the growing performance requirements [ITR]. Through this, an equivalent amount of operations per second can be achieved at far less time than in a purely sequential processing. The parallelization of the workload can be achieved on different levels of granularity (instruction-level parallelism, data-level parallelism, task-level parallelism), and the achievable gain largely depends on the application itself. For example, a digital filter application typically exhibits significant instruction-level parallelism, while a heterogeneous application such as a user interface offers only parallel tasks. Commonly, different levels of parallelism support can be provided by an architecture.

Multicore Processors Multicore controllers (also called multicore processors), provide an efficient means to supply additional performance by combining several benefits. Firstly, they are fully programmable, enabling a relatively easy functional port of existing single-core applications. The architecture provides task-level parallelism that is very easy to exploit, especially if formerly independent functions are to be integrated. Secondly, the comparably low clock frequency makes multicores power-efficient, avoids heat problems, and provides a resistance to electromagnetic interference (and limits the emitted radiation). Thirdly, the independent cores offer the option for independent scheduling, which opens the way to physical redundancy as well as an evolutionary path to the high-performance applications through scheduling partitions and operating system virtualization. These benefits make multicore processors an attractive design target in across virtually all computing domains.

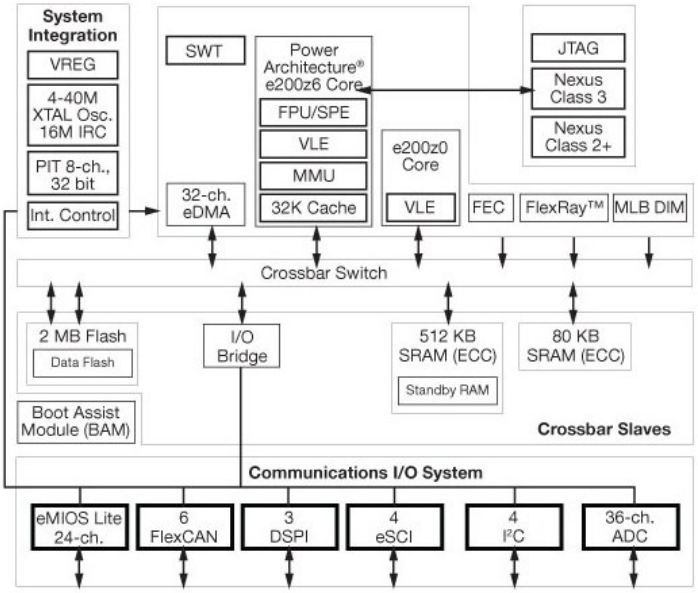

Figure 1.2: Block Diagram of Freescale MPC5668G Dual-Core Controller for Gateway Applications [Fre09b].

The block diagram of an automotive multicore controller is depicted in Figure 1.2. The controller consists of two (heterogeneous) processing cores, one larger core is dedicated to sophisticated computations, and the smaller one is used to offload common packet pre-processing operations. The large core is equipped with a unified 32KByte cache and the hardware units to support a full scale operating system (such as the Autosar OS [AUT09]). Both cores can access a variety of resources (such as 3 dedicated memory units, and a bridge to external I/O) via memory mapped bus operations. The shared resources can additionally be used by other master devices (such as the FlexRay controller), which for example may write into the same memory. This common use of the resources causes the timing and performance of the components to be entangled; which can be easily overlooked during verification — it will be shown in Section 1.4.1 that this can be the root for severe timing problems.

The availability of multiple processors also introduces an additional dimension of scheduling freedom: Tasks can not only be assigned over time, but possibly also over different cores. The evolutionary road that is taken in the automotive industry is to resort to a partitioned scheduling at least in the foreseeable future. This is motivated by the greatest compatibility to present designs and the observations that

the run-time migration of task between cores during run-time entails a significant overhead. Furthermore, research has already shown that migratory scheduling does not necessarily provide better real-time performance [Bak05].

The more such parallelization approaches are used in safety-critical applications, the more one has to ask whether the estimated performance can be guaranteed also in critical scenarios. One may tolerate timing glitches in a desktop PC, or even in a car infotainment system. But when the engine control loses data, this can result in higher fuel consumption, broken valves, stopped engines, and even pose a risk to the passenger. Consequently, methods and tools are required to ensure the safe design and application of these systems.

The following section investigates the development process in embedded systems, and Section 1.4 then identifies key challenges to the predictability of such systems by the upcoming hardware and software architectures. We then discuss previously proposed countermeasures in Section 1.5 and highlight the motivation and benefits of our solution in 1.6.

1.3 Development of Embedded Real-Time Systems

To conquer the increasing complexity of today's embedded systems, sophisticated engineering methods are required that fit the designers with the necessary information to find an optimal design and provide them with the leverage to introduce modifications into the process. According to the position paper [SV07], key technical challenges to successful development of future embedded systems, are the heterogeneity and complexity of the hardware platform, the growing complexity of the embedded software, and the integration of the subsystems. In practice, the solution is a combination of a controlled but flexible development process in combination with design tools that assist and automate the intermediate steps.

The development process in the automotive industry, especially in Germany, is traditionally centered around the so-called "V-Modell"[Drö98], in which the development is separated into a "specification and design" branch and an "implementation and integration" phase along the horizontal (time) axis, and the suggested level of refinement and abstraction along the vertical axis. This is shown in Figure 1.3. The specification branch begins with the specification of the system's requirements that define the system functionality and interaction with the physical world. These requirements are the basis of the system design, that in the next step will be broken down into the design of several components. The component functionality is then implemented largely independently (possibly based on hierarchical application of this development process). Once the implementation is complete, the components have to be integrated to comprise the complete system. The key idea is that each level of abstraction can be individually tested against its specification, and not only at the system level integration, when the cost of redesign becomes very high.

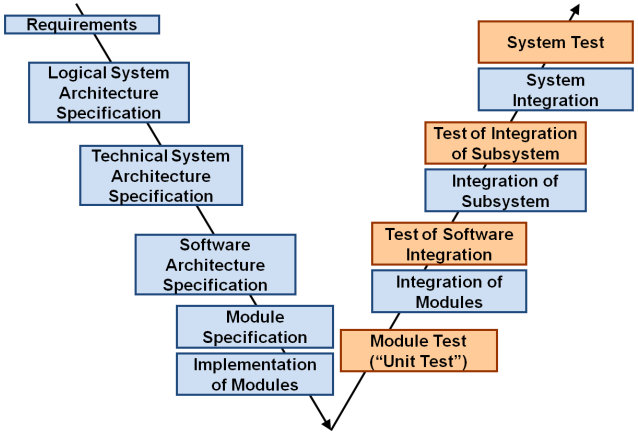

Figure 1.3: Development Process according to "V model"[Drö98]. Phase I: Specification and Implementation, Phase II: Integration and Verification

OEMs (the original equipment manufacturers) face two major challenges in this development process. Firstly, fundamental architectural decisions have to be made relatively early. For this, estimates on the performance of different implementation scenarios must be evaluated, even before the actual implementation has begun. Secondly, the OEM will integrate the components, which are commonly developed by different suppliers, into the subsystem. It has become evident in recent years that the common use of finite resources such as buses, memories, or coprocessors, introduces integration risks [Ric04], which we will highlight in Section 1.4. To be able to address these risks, the timing and resource requirements have to be an explicit part of the specification.

The goal is therefore to accompany the functionally oriented development process with timing specification and verification steps. During the early (specification) phases of the design, the system architecture has to be defined, and its resources broken down into specifications for the different subsystems. This step is critical, yet challenging, because no reliable data is available about the performance requirements of the individual applications. Therefore, the process is usually based on estimates and data available from previous product generations. During the integration phase, the actual resource requirements per application then have to be verified, and their concurrent behavior secured.

In the following subsections we review the established methods to consider timing different phases of the development process. Then, in Section 1.4 we identify the challenges imposed on this process in light of the changing new hardware and software

architectures.

1.3.1 Specification and Implementation Phase

The timing concerns in the development process can be expressed with so called timing specification languages that allow to assign time budgets and constraints to various functions (e.g. TADL (timing-aware description language) [TAD09, Jer07]). An important concept here is the notion of "events" that denote task or message activations and completions. The marked conditions can be refined for different abstraction levels (e.g. "vehicle-level" events down to "operational-level" events). Such annotation languages are agnostic to the actual implementation language.

Also some specification languages are immediately concerned with specifying the timing behavior. For example the synchronous languages [Ben91] assume that the computation of results, and their propagation to the outside take place in distinct, non-overlapping phases. The properties of the developed application than hold independently of the actual implementation (or mapping of the function), as long as it is faster than a certain threshold.

When specifying the behavior of the system, it is key to choose a suitable model of computation in order to allow for an efficient implementation. The diversity of operations in some high level languages (for example in C) can quickly sacrifice the analyzability [God07, SV07]. Therefore, many specialized models of computation have been proposed that allow providing a mathematically complete specification from which the actual implementation can be derived — ideally automatically through synthesis [Edw97] — and which allow tracking the behavior with formal methods. There is a large body of work on the combined application of different models of computation [Eke03].

A similar efficiency impact is implied by the choice of the run-time organization and control of the inter-task communication, in particular in multicore systems (also called "programming model", as discussed in [Har08, Pop09]). Different general paradigms can be distinguished: the symmetric multiprocessor organization, in which the workload can be dynamically assigned to each processing node, the client-server oriented architecture, in which tasks can request services through messages and responses, and the streaming approach, in which the application topology is fixed and the efficient flow of data is the dominant concern. A programming model consists of the corresponding inter-task communication primitives, task and resource control primitives (i.e. scheduling hooks), and hardware access primitives (e.g. for I/O), and should ideally allow a refinement along different abstraction levels. In the end, the hardware architecture and application topology impacts the efficiency of the programming model [Pau06, Pau09].

1.3.2 Integration and Verification Phase

After the system components have been implemented, the integration process begins. During the integration, one has to make sure that the assumptions made during the development of the subsystems actually hold (e.g. that they receive sufficient resources), and that the complete system behaves according to the specifications.

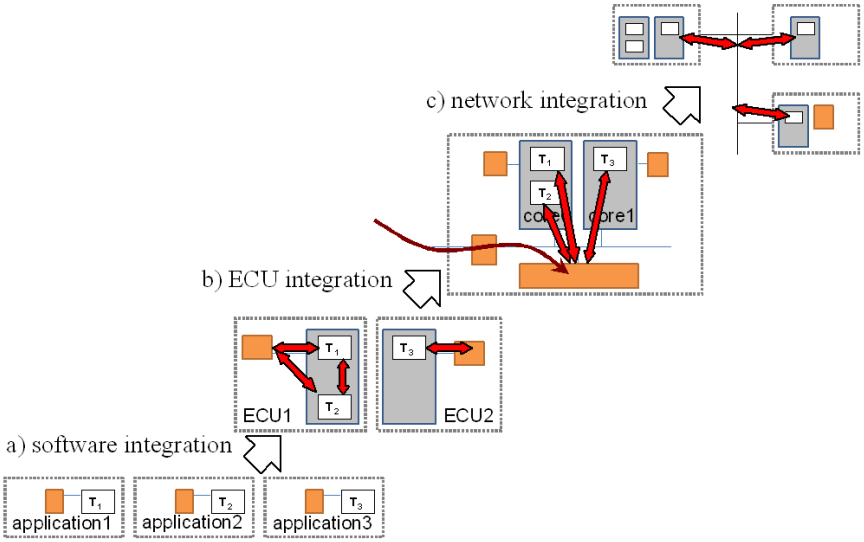

Figure 1.4: Steps in the Integration Process ("software integration": integration of different applications on the same processor, "ECU integration": multiple processing components per ECU, "network integration": integration on joint communication infrastructure).

Rarely will a software function be implemented by a single task (or runnable) alone. Rather, the functionality will be distributed on multiple processing nodes for reasons of component specialization, data locality, and workload parallelization. A key phase of the integration process is the "network integration", in which these applications are assembled into the same system, using the same communication infrastructure as shown in Figure 1.4 c). In this phase the resource and timing requirements from the different applications must be reconciled with each other. Any failure to correctly specify the application's communication behavior and accurately predict the joint execution will lead to a resource misdimensioning (with the known implications for cost and safety).

A growing importance comes to another level of integration, the "software integra-

tion", as software is increasingly provided by different suppliers (Figure 1.4 a)). The AUTOSAR consortium, a gathering of most active players in the automotive domain [aut06] is working on simplifying the division of labor not only per ECU, but also per application. For this, interface standards have been developed that specify a runtime environment (RTE), a basic software (BSW), and the user functions (Software Components), that communicate over a virtual function bus (VFB). This hardware abstraction is aimed to enable the late mapping of the software components to the actual processors, where they can possibly co-exist with software supplied by a different vendor (see [Bro07] for more information on automotive software engineering). Thus, the tasks from independent applications will now compete for the same processor. In addition to the functional issues (that are relatively well covered by AUTOSAR), this introduces timing issues that have to be closely investigated.

A third level of integration is introduced by multicore components. Applications can now have timing dependencies even if they are mapped to different cores (Figure 1.4 b)). Again, the functional isolation is relatively easy to achieve (for example through virtualization), but the use of shared resources introduces challenging new timing problems (which will be illustrated in Section 1.4.1).

Thus, the problems known from "network integration" will in the future be observed much earlier during "software integration" and "component" or "ECU integration". In the next section, we will highlight some of the key challenges.

1.4 Integration Challenges

The system setups to be observed are thus of a challenging complexity: Firstly, the applications consist of multiple tasks that are distributed to multiple control units in the system. Each control unit can service the tasks of multiple applications. For this, it is either equipped with a sufficiently powerful processor, or due to the reasons cited in Section 1.2.2, multiple processors and coprocessors.

The competing applications are usually scheduled with dynamic scheduling policies that assign the resource according to the run-time execution requirements of the individual tasks. This leads to an efficient utilization of the available resource, but also introduces dynamism into the run-time behavior that is challenging to predict at design time. As highlighted in [Ern03, Hen05, Rac08], the timing of such systems is usually not even free of anomalies, i.e. if the output of one task is produced earlier than expected this can lead to another task not being able to finish on time. If left undiscovered, such a behavior poses a tremendous risk to the safe application of the system.

Formal analysis concepts are ideal to defuse these integration risks. The integration process can be covered by a matching bottom-up analysis as discussed in [Ric06]. We will look into present approaches to this problem in Section 1.5.3. Usually, there is a strong distinction between a per-processor and a system-level scope. This layered approach can however not be maintained when multicore control units are applied.

1.4.1 Multicore Integration Challenges

We have seen in Section 1.2 that multicore processors can be expected to play an increasingly important role in automotive and other control-dominated domains. In addition to the integration risks known from distributed systems, multicore systems pose novel challenges to the established development process for embedded systems, and automotive systems in particular.

The strong spacial locality of the processing components makes the sharing of chip-level resources very attractive, and consequently operations on shared resources are becoming a common part of embedded applications. The use of a shared resource during the execution of a task introduces a second level of arbitration beyond the local processor: A task can only make progress when it has both (or alternatingly) been granted local execution time on its processor, and access to the secondary shared resource. From a design perspective, this leads to unwanted side effects, such as a timing interdependency between different tasks on different cores: A low priority task that makes excessive use of the shared resource possibly slows down a high priority task on another core. In [Wil09], for example, this is one of the reasons why multicore has been identified as a key challenge to timing predictability and verification.

Example To illustrate the timing implications of shared resources consider the following example. Assume a dual-core embedded control unit that executes 2 periodic and one event-driven hard real-time task. The tasks are statically assigned to the processors, and scheduled according to (partitioned) static priority preemptive scheduling. During execution, the tasks arbitrarily read data from a memory. Assume that the memory can only serve one request at a time and that a core is stalled whenever a task performs a memory access until that request has been served.

Possible resulting run-time schedules are depicted in Figure 1.5. In the first scenario, Scenario 1.5a), the memory is always available for the tasks on each processor. This is the behavior known from single-processor setups. The response time of the lower priority task contains phases of execution, preemption, and processor stalls due to the memory accesses (which are contributed by both tasks, as in either case the low priority task can not execute). In effect, the low priority task is in this case kept from executing by three invocations of the high priority task.

Scenario 1.5b) shows the case where the memory is also used by a task on another processor, in this case periodically. Whenever the memory is also used by a task on core 1, core 0 is stalled for a longer amount of time, which increases the task response times. In addition, the response time of the low priority task has grown so much that it suffers a fourth preemption by the high priority task. Thus, there may even be a super-linear increase of the worst-case response time. These effects challenge the safety of the task's deadlines. Critically, the interference can affect tasks across all priority levels; by using the shared resource, a very low priority task on one core can impair the performance of high priority tasks on another.

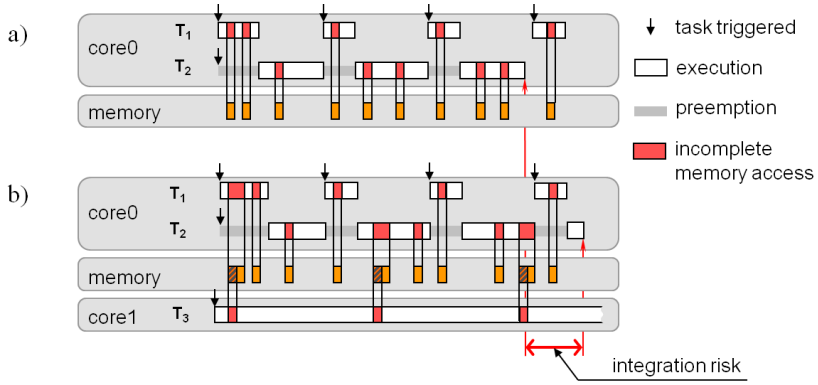

Figure 1.5: a) Single Processor Accessing a Memory b) Conflicting Accesses from Another Processor.

Typical sources of such timing interdependence are the physical main memory or coprocessor, or logical data structures protected by semaphores. The accesses can be explicit, thus specified through special instructions in the application binary, or implicit, when they surface only during run-time. For example, executing code from a cached memory will cause implicit cache misses that are usually highly dynamic with respect to their occurrence and timing during run-time — and therefore also difficult to predict at design time.

Thus, the task interdependence known from scheduling and processor sharing can now also be observed across cores, which challenges the previously established integration and analysis procedure. In a distributed system without shared resources the performance analysis could be performed bottom-up (as suggested in [Cha03a, Ric02b]), by first deriving a task's execution time based on processor properties (such as the instruction set and pipeline length), and second the task's response time based on the execution times of the tasks sharing the same processor. Due to the inter-core influences, this procedure is not feasible. The competition for the shared resource — a "system level" operation — feeds back into the execution of a task, making previously calculated values such as the worst-case execution time invalid. Thus, task verification can not be performed in isolation, not even if the tasks are mapped on different cores.

1.5 Tackling the Integration Challenges

As the integration problems in multiprocessor and multicore systems become evident and more grave with respect to growing system complexity, a number of different approaches have been suggested to ameliorate the situation.

1.5.1 Traditional Approaches

Simulation has long been the predominant method of choice in industry to investigate a system's behavior. It is relatively easy to tailor a simulation framework to a new system, and it can be executed on different levels of abstraction, either including or excluding functional aspects. Virtual prototyping allows moving the functional (and some non-functional) exploration and verification earlier during the product development [Val97, Sch08d]. As investigated in [Sch08a], the choice of modeled details and abstraction then implies a trade-off between speed and achievable accuracy.

Any such investigation will be based on a set of input stimuli that aim to cover the behavioral space of the system. However, this space quickly becomes very large — which leads to large run-times; and the behavioral dependencies are often intricate — which makes it infeasible to find an exhaustive set of stimuli. Consequently, simulation-based approaches carry the risk of not covering critical scenarios, leaving the integrator with a risk of undiscovered violations of the specification.

Trivially also, the timing impact of shared resource usage can be avoided if the resource is not shared. Any reduction in the need to occupy and use the shared resources will lead to a system that is easier to investigate, and that potentially has a better performance, because every shared resource access requires some synchronization between tasks that reduces their effective work time [Cul10]. However, there are several reasons why this is not universally applicable: Firstly, in a typical multicore system design (such as shown in Figure 1.2) many resources will not be available "per core", but rather "per chip". For example, there will be one shared controller to the external system bus, one central interrupt controller, one shared DMA unit. This makes a certain amount of shared resource accesses unavoidable.

Often also the memory is shared. In this case a second point implies heavy use of a shared resource. When no shared memory exists, tasks on different cores share data by sending each other messages over a bus. The larger the data, the more time and resource consuming this becomes. In the presence of a shared memory, the messaging operation consists simply of sharing a pointer between tasks on different cores. Besides this efficient data sharing, a shared memory can also be better utilized because there is less fragmentation loss in the overall memory allocation.

In light of the limited applicability of these traditional methods to the integration challenges, industry and academia is turning to new approaches. These procedures, namely the orthogonalization of shared resources and the investigation of the run-time interference with formal analysis, are illuminated in the following sections.

1.5.2 Time-Orthogonalization of Resources

The integration of modules with a competing resource requirements can be largely simplified by resorting to an *orthogonalization* of the resources [Lee05]. In such an approach, any dynamic run-time interference between the different applications is to be

prevented. This can be achieved for example by assigning the processors, buses, and shared resources to the applications according to a predefined time or budget. The industry is investigating the use of time-triggered architectures to ease the system-level integration process by introducing network protocols like FlexRay [Fle06] in automotive, TT-Ethernet [Kop05] in avionics, or Profinet [Fer06] in industrial automation. Similar considerations are carried out with respect to scheduling of the processors (see [Aer03] in avionics).

The problem with this approach is that it generally introduces an overly conservative design, because the available budget has to be partitioned among the applications, making it difficult to efficiently reuse resources that are released early. Furthermore, any design robustness towards future modifications and additions has to be a priori considered (as suggested e.g. in [Gho10]). Additional hardware measures can be applied to protect the system against failing subsystems that over-use their assigned budgets. This makes this approach interesting mostly for highly safety critical applications with little post-production updating and long product cycles.

Usually this high level of safety and strict partitioning is often required only for a subset of the executed applications. For the remaining workload, it is desirable to achieve better flexibility and efficiency. Hierarchical scheduling approaches promise to combine the benefits. FlexRay for example provides a time-triggered and a dynamically scheduled segment for this purpose.

It is important for an efficient design that the unused time slots in a time-triggered design can be recouped by other tasks in the system. This is for example provided by round-robin scheduling, that ensures an upper bound on the interference per task, and an implicit re-assignment of time slots that are waived. Recently, this effect has been studied also for real-time workloads (in [Rac07]), by including the guaranteed amount of waived processing time. The unused service can also be redistributed according to priorities [Ste09]. Unfortunately, analyses often neglect the guaranteed recuperation that is present also in the worst-case (i.e. in the worst case it is usually assumed that not more than the predefined budget is supplied, although the task may receive more service at run-time).

Orthogonal timing between resources in a processing time can also be achieved by implementing the communication between tasks not via queues but via registers that are read (and written) non-blocking at regular intervals (as suggested in [Ben07]). The problem with this approach is that the output of the system then depends on its timing: if data is read too late, it might already be overwritten, which can lead to a different output. In this case, one has to make sure that the age of the data is within the bounds required by the application [Fei08]. For the scope of this thesis, we focus on tasks that communicate via FIFO buffers, which ensure data consistency (as shown in [Wig09]). However, a combined application of both communication methods is also conceivable. For this, the segmented timing chains suggested for the industrial AUTOSAR standard are a suitable starting point [Jer07, Esp08, Klo10].

Orthogonalization concepts can also be applied to ameliorate the feedback effect of secondary shared resource timing on the task execution in multicore components (as suggested in [Wil09, Pus08]). This can be achieved for example by using a system crossbar or arbitrating the resource assignment through time-driven scheduling of the memory bus (as in [Pau02][Bek04]). By reducing the timing interdependence, the tasks on each core can then be verified separately. But, also here the issue is a potential over-dimensioning of resources: Typical problems are the treatment of load requirements that vary over time, or the synchronization between elements of a processing chain. We will review the work related to this problem in more detail in Section 5.2.

To ease the functional aspects of a transition from a distributed multiprocessor environment to integrated multicore units, *virtualization* is often used. In such a setup, a hypervisor task will intercept certain hardware operations, and through this provide e.g. an independent memory address space to each partition. Using virtualization techniques, each core may feature a unique operating system and scheduler, and data between the cores is shared e.g. via loopback bus devices. Applications developed for single-core components can then be easily retargeted — given that timing is not a concern. However, as the resources are still physically shared, the timing problems generated by the inter-task and inter-core competition remain.

1.5.3 Formal Performance Analysis

Due to the unavailability of orthogonalization measures in common multiprocessor systems, and the insufficient coverage af simulation-based approaches, formal analysis is an increasingly important method to safeguard the dimensioning and integration of sub-system implementations. In this section, we give an overview over representative methods and approaches. We will roughly classify the approaches according to the level of abstraction at which they are used predominantly, although there certainly is some overlap (especially for meta-methods such as model checking). Also the present thesis is concerned with formal performance analysis, so as far as there is a relation to a particular problem addressed in this thesis, additional related work will be highlighted in the respective chapters.

1.5.3.1 Basic Task Analysis

Formal analysis is traditionally applied at different levels of abstraction that follow the integration and verification phases shown in Figure 1.4. The first concern is the derivation of the *execution timing of tasks* that execute on a specific processor (with a given architecture). These metrics can be derived for each task in isolation with static analysis of the task's control flow which contains the logical structure of the task execution. A comprehensive overview of such analysis techniques can be found in [Wil08]. We review the relevant methods and adopt them in Chapter 5 and again in Section 8.1.

Alternatively, for the limited scope of single task analysis, extensive simulation is also used in practice, but at the risk of missing critical corner cases, i.e. not finding the longest path [Giu01, Col03].

1.5.3.2 Formal Analysis for Processor-Level Integration

On the basis of the individual task timing and its external activation behavior, the *processor timing* can be investigated. The most important metrics in real-time systems are the task's best- and worst-case response time. This simple metric has long been the focus of research in single processor scheduling theory [Liu73, Jos86, Leh90, Tin94b, Bur95]. Many analyses are based on the *busy-window approach* that accumulates the delay that a task may experience during its response time.

Variations and extensions of this technique have been proposed, which improve the analysis results by considering more detailed task models, such as offsets [Pal98] or variable task execution times [Mok97]. Also, realistic scheduling effects, such as cache-related preemption delay in preemptive scheduling [Sta05b] or specific protocols, such as the FlexRay bus protocol [Pop06], can be covered. We refer to these approaches again in Chapters 3 and 6.

1.5.3.3 Formal Analysis for System Level Integration

Finally, the system-level integration phase requires a global view and analysis of the system, taking into account the various interdependencies between task scheduling and communication.

Formal performance analysis can be applied in different phases of the embedded system design presented in Section 1.3. As previously suggested, it can be used for verification of the integration steps along the integration phase, and ultimately to assist the certification of the product development. But it can often be used much earlier due to the abstract model: It does not require a detailed platform model or executable code in order to predict the performance. Thus, based on estimates and extrapolations from previous product generations, one can establish a timing model of the system relatively early and use this model to guide the development process towards the most efficient solutions ("design space exploration" as in [Thi02, Ham06]).

One method to investigate the behavior and timing of a system is by modeling it as a *timed automata* as in [Eri98, Hen06a, Bre08, Dav09]. Here, a formal model of the system timing is subjected to a *model checker* that verifies the adherence to timing properties through a reachability analysis [Bai08]. Model checking allows to specify a multitude of global dependencies in order to deliver tight performance bounds. However, model checkers rely on an exhaustive state space coverage, which does not scale well with system size and heterogeneity. Therefore, it is difficult to apply these approaches to arbitrary systems. It can, however, be used to investigate an isolated subsystem.

To avoid this problem, the analysis can take a more problem-aware approach by recognizing inherent dependencies and reducing the analysis effort. For this, the single-processor scheduling theory was systematically extended towards specific combinations of input event models, resource sharing and communication policies in [Tin94c, Gut97, Har01, Pop03]. The problem with this, sometimes called "holistic", approach to system level analysis is that it has to be adopted to each specific setup, making this approach inappropriate for today's versatile designs.

This adaptation problem is avoided in the *compositional* or *modular performance analysis* approaches [Ric03, Cha03a, Hen06c], which break down the complexity of the complete system into separate local component analyses, and a procedure to derive the combined system properties. In [Cru91b, Gre93a, Sti98, LeB01, Ric03, Cha03a], the key idea is the application and derivation of *event models* that contain the relevant information about the flow of events and the corresponding workload imposed on the resources. Such approaches are key to address systems with chained task activations, as they allow to deduce the event flow at a task's output from the task activating event model and a description of the service provided by the resource. Unfortunately, as we will see in Chapter 3, previous methods are either constrained to specific classes of event models, or imply relatively complex operations that lead to large analysis times.

An event-driven system may even exhibit cyclic dependencies routing from chained and data-driven task activations in combination with dynamic scheduling. These dependencies can often not be avoided due to legacy implementations or because they represent the only schedulable solution. Then, no sequence of analyses exists that allows a straight-forward procedure. To tackle this problem, compositional performance analyses can be combined with fixed-point theory to iteratively solve the analysis dependencies [Ric03, Thi06, Jon08, Ste11]. This procedure exhibits great flexibility and scalability for timing and performance analysis also of complex distributed embedded real-time systems. We adopt such a procedure, and will thus hear more about these concepts in Chapter 2.

Other approaches derive the system's behavior not by composing the corresponding analyses, but by *parallel composition* of the components' behaviors. In [Lee98], the combined behavior of two composed components is the intersection of the corresponding individual behaviors. This approach has been applied also towards systems that are modeled with heterogeneous model of computation assumptions [Ben08]. This analysis process again suffers from potentially intractable sets of behaviors that need to be considered. Therefore, the approach relies on the presence of time-triggered (budgeted) architectures, for which conditions for correctness-by-construction can be identified [Ben04].

The timing behavior of a system can also be modeled with the help of *dataflow models* [Sri00], such as synchronous dataflow graphs [Lee87] or cyclo-static dataflow graphs [Bil96], which represent constrained models of computation with strong mono-

tonicity properties. System properties such as the buffer requirements or the throughput of a system can then be derived on the basis of *max-plus algebra* [Bek04, Bac92]. While the approach delivers accurate results for static systems and predictable arbitration policies, it is not suitable to address schedulers in which the guaranteed service supplied to one application depends on the run-time load imposed by another (see also Section 3.4).

The large spectrum of analysis approaches shows that there are different trade-offs to consider, mainly between the expressiveness of the model, the analysis speed and complexity, the analysis accuracy, its delivered state coverage, and in practice also the ease of its application (see also [Per08]). To exploit the individual strengths of the approaches, different methods can also be combined. This has been shown for example for simulation and formal analysis in [Kün06, Sch08e, Sch08b], symbolic performance analysis and timed automata in [Lam09], symbolic performance analysis and dataflow-based analysis in [Sch07, Thi09], or different facets of the symbolic performance analysis in [Kün07].

The importance of considering non-functional design aspects in system-level design is today also acknowledged by the industry. For example, an appropriate timing extension for AUTOSAR is currently under development. In parallel, the TIMMO project, with OEMs, tier-1 suppliers, and tool vendors develops a formal language and a methodology for timing and performance design for a range of automotive domains [Jer07]. These modeling standards provide the basic ground for the serious application of formal methods. Thus, methods that have been suggested in research for some time, now become feasible to be used in actual productive environments.

1.5.3.4 Formal Analysis Considering Multicore

The above classification into task-, processor- and system-level analysis approaches neglects the component-level abstraction that, as laid out in Section 1.3.2, is of growing importance in multi-node cyber-physical systems. The tight coupling of the involved tasks and resources provides new opportunities for an efficient design, but also requires additional attention to uncover the timing interdependencies and resulting hazards. The heavy interaction and possibly large number of shared resource accesses by the co-running tasks make the timing characteristics at this level of integration distinctly different from classical processor-level combined with system-level interactions.

The new dimension of scheduling freedom provided by multicore processors allows assigning tasks not only over time but also over different cores. If the tasks can be migrated between cores at run-time, we talk about *global* scheduling policies (as opposed to *partitioned*). Feasibility tests have been developed for a variety of global scheduling policies [And01, And03, Bar08b], some even considering arbitrary task activation patterns (as opposed to periodic) [Bar07]. Recently, the busy window technique known from uniprocessor scheduling analysis has also become feasible in

this setup [Ber07, Gua09].

Partitioned scheduling better matches the classical abstraction levels, and is therefore the evolutionary approach to harness the computing power of multicore components. But in the presence of dynamic arbitration of shared resources, the analysis is faced with several new challenges even in this setup.

The manner in which the shared resources are accessed is fundamentally different from the classical task chaining model that is often assumed in the system-level analysis of distributed real-time systems. Instead of individual task activations triggering each other, a task instance may require anything from a few to several thousands of shared resource operations (in particular when the shared resource is a memory). This inherent multi-rate relationship complicates performance analysis due to the possibly exponential nature of the resulting states (see e.g. [Sri00, Gei09]). This makes the direct adoption of the present system-level approaches inappropriate. Novel ideas to address this challenge are discussed in the related work section of Chapter 5.

Moreover, there is a mutual dependency between the analyses of the involved processing cores. The example in Section 1.4.1 has shown that the duration of shared resource access delay that can be experienced by a task depends on the amount of traffic imposed on the shared resources by other processors or processing elements in the system. The respective local analysis of the other processors however also requires the knowledge of the shared resource delay. This closes a cycle: The timing interference between the components of this setup translates into mutual dependencies between the local analyses of the different cores. As discussed above, a similar dependency problem has been tackled for distributed systems with compositional performance analysis and fixed-point theory, which we revisit and extend in Chapter 2.

To facilitate the analysis, new models are required to approach the various new timing parameters of such systems. Key metrics are the load imposed by a task in a shared resource, the worst-case delay of a large set of shared resource operations, and its impact on the task's worst-case response time have all only received only marginal attention by previous research.

1.6 Summary and Contributions

1.6.1 Summary and Problem Statement

This chapter has laid out why there is a growing demand for computing power in embedded systems across various domains, and that this demand leads to significant challenges to the development process, in particular for a successful subsystem integration. While previously, integration in the automotive domain was performed on the level of network nodes, it will in the future also take place within the ECUs. Thus, applications consist of multiple event- and time-driven tasks that are executed on hierarchical platforms: tightly coupled task-systems on single- and multicore processors are embedded components of a networked system.

Besides demanding for conservative performance guarantees that allow safeguarding the deployment of such systems in safety critical environments, the cost-sensitivity of high-volume markets such as automotive requires formal analyses to reflect the actual behavior as accurately as possible. But with growing system complexity, this requirement is becoming difficult to fulfill. With respect to timing, the designer is looking for realistic event models, accurate response times, and tight end-to-end latencies.

During the multicore era, a designer will additionally be concerned about novel performance metrics that have not received sufficient attention by the research community: One has to establish conservative metrics and analyses for the load on the shared resources, the shared resource access delays, and quantify its impact on the task's response times. These metrics are particularly important in the presence of shared memories and caches, yet this scenario is particularly difficult to investigate due to the dynamic behavior.

The use of shared resources introduces various mutual dependencies that complicate the analysis process: Firstly, there is the cyclic dependency between the worst-case response time of tasks on different processors, so that neither can be computed without the other. Secondly, dynamic online scheduling increases the run-time dynamism of the shared resource requests. And thirdly, tasks are not always activated by external events that can be characterized in advance, but rather tasks can be chained, thus the activation of one task is the effect of another task or communication being finished. The complexity of the resulting analysis should be hidden from the designer as much as possible, so that she or he can focus on optimizing the projected behavior.

1.6.2 Contributions

The benefit of formal analysis for the design and verification of distributed multiprocessor systems has been demonstrated in numerous academic and industrial examples. But as seen above, there are a number of open issues to consider before its potential can be leveraged also in hierarchical system setups that consist of or contain multicore components. The approach proposed in this thesis aims to address these problems and provides the following key benefits:

- *Support Event-driven Systems and Dynamic Run-Time Scheduling.* The proposed approach recognizes the fact that many systems, also in the control domain, are inherently event-driven and scheduled dynamically according to the run-time workload. This causes a multitude of dependencies between the task activations, their shared resource usage, and their response times. We will show in this thesis that the relevant behavioral intervals of these properties are actually co-monotonic. With this in mind, we provide an iterative analysis approach that tackles the dependencies through fixed-point theory.
- *Support Run-time Arbitration of Shared Resources.* The thesis addresses the key challenges of the multicore integration, where resources are dynamically

shared across the cores, often in the absence of protective measures such as time-orthogonalization. This is possible by adopting the event model concept — known from modeling task activations in distributed systems — to express also the load imposed on the shared resources. This measure allows breaking down the analysis complexity into three distinct steps: We provide new analyses to determine the possible load of single or multiple preemptively scheduled tasks, identify the latency of the shared resource accesses, and quantify the contribution of these delays to the task's response times.

- *Improved Accuracy through new Models* The timing of individual tasks and resources is in this thesis captured with novel, more expressive models, namely the aggregate busy time function, and the multiple-event busy time model. This adequate resource abstraction enables the derivation of key performance metrics more precisely than previous approaches — without introducing the complexity of the operations on service curves as known from network calculus. We show how the timing of task activating events inside the system can be derived from arbitrary load event models. This procedure accurately captures also irregular event streams and the distortion of event streams through scheduling variability. The new timing models are also used to provide an new end-to-end latency that considers the pipelined processing of events, correlated timing in fork-join application structures, and even cyclic dependencies.
- *Decomposition of Analyses Concerns.* Using this methodology, the local scheduling analyses per processor and the analyses for the shared resource arbitration are decoupled. This allows for the combination of heterogeneous components that can be replaced, refined, and optimized independently without endangering the analyzability of the complete system. Furthermore, specialized methods can be used for the investigation of each component, which increases the achievable accuracy.
- *Seamless Integration into established System-level Analysis Process.* The analysis of multicore components as presented in this thesis is conceived to be fully compatible with the established system-level compositional analysis approach based on [Ric03].

1.7 Outline

The thesis is organized as follows. In Chapter 2, we lay out a versatile compositional approach that enables the decomposition of the analysis problem into the dedicated analyses of the various system parameters. This ensures that the successively introduced analysis modules can actually be integrated to derive conservative system properties.

Our focus will then initially be on the interaction between tasks in a system consisting of multiple independently scheduled processing nodes. We investigate the timing of event-driven activations in task chains (in Chapter 3), and end-to-end latencies over

multiple tasks (in Chapter 4). For both we rely on novel task and event models to attain efficiency and accuracy.

We then turn to the problem of shared resources in multicore components that present an additional analysis complexity. Novel analyses are introduced for three key metrics to investigate such nodes: First, we provide bounds on the run-time load that can be imposed by tasks and sets of tasks on shared resources (in Chapter 5). Then, we investigate how the shared resource delays contribute to the task's response time and how they can be included in the analysis (in Chapter 6). Finally, we turn to the problem of deriving the total duration of the shared resource operations considering the run-time arbitration (in Chapter 7). Each analysis component stands on its own and provides valuable performance metrics in case all other parameters are known, but in addition we show that they comply with the conditions to integrate them into a joint system level analysis.

In Chapter 8, we present applications of the provided analysis framework. In one application, we stretch the analysis scope to cover the challenging dynamic timing of a shared memory that is accessed via local instruction caches. In another, we investigate an industrial multiprocessor platform running a multimedia application.

Finally, we summarize our findings in Chapter 9 and draw a conclusion.

2 Compositional Performance Analysis of Systems with Shared Resources

Given the heterogeneity of typical embedded multiprocessor system designs, it is often impossible to find a closed-form equation that directly allows deriving the internal timing parameters. Many systems feature heterogeneous components that service multiple applications at the same time. Such a setup quickly becomes too complex to capture holistically. Instead, the problem has to be broken down into feasible sub-problems and a procedure to integrate the results for the complete system. Such a hierarchical analysis procedure has already been demonstrated to work for distributed systems [Gre93a, LeB01, Ric02b, Cha03a]. There, the most relevant analysis parameters are the task activating event models.

We adopt this concept and generalize it to encompass also other timing parameters, such as the delays incurred by the use of shared resources. In this model, each relevant parameter either has to be provided by the designer or an analysis function has to available that allows its derivation from the remaining parameters. Dependencies between the analysis functions will be resolved by iteration.

This procedure has several strong advantages over holistic approaches: First of all, it breaks down the analysis complexity, because not the complete set of interdependencies and analysis equations needs to be considered at the same time. Rather, the scope for each sub-analysis is local and tractable. Secondly, the dissection allows the application of specialized solving and analysis techniques, which can be tailored for a specific sub-problem. Thirdly, through modularizing the analysis, the techniques for specific sub-problems (such as the timing properties of a specific operating system) can be reused, when an equivalent scheduler or arbiter is applied in a completely different system.

In this chapter, we first survey related work in Section 2.1 and outline the proposed analysis and introduce basic model properties in Section 2.2. We then show how such an analysis can be decomposed into multiple sub-analyses in Section 2.3. This allows us to identify the key parameters of the investigated class of systems in Section 2.4. Finally, we provide a procedure to integrate these results in order to derive conservative system properties in Section 2.5. A key idea is to explore the established methodology of “fixed-point theory” in Section 2.5.3 to iteratively derive the unknown properties of the system. We conclude the chapter with a short summary in Section 2.6.

2.1 Related Work

The basic idea of decomposing the performance analysis of a multiprocessor system into local analyses was first applied in [Gre93b, Gre93a], where an event-stream model was introduced and used to couple the analysis of independently scheduled components. Analysis composition is also the driving idea behind *network calculus* [Cru91a, Cru91b, LeB01] (see [Fid10] for a recent survey on the related methods), that allows reasoning about the flow of events over multiple resources. The event model derivation process was also adopted in the "compositional performance analysis" of [Ric02b], where it was combined with transformation functions to allow composing heterogeneous schedulers and in *real-time calculus* [Thi00, Cha03a], which contributes refined procedures on the derivation of the resulting service provided in real-time scheduling.

These approaches are compositional in the sense that they separate the problem into local component analyses and the modeling of event flow between them, but they can not be directly applied to problems in which functional or non-functional timing dependencies exist between the components. For such cases, the use of fixed-point theory was suggested in [Ric02b, Ric03], and adopted to other approaches as in [Thi06].

The basic idea of decomposing the performance analysis for multi-node multiprocessor systems is as follows. The procedure interleaves the analysis of individual resources with the propagation of event models (depicted in Figure 2.1). First the environmental input event models representing the minimum and maximum amount of events that the system is exposed to are specified (1). All other input event models within the system are initialized with optimistic assumptions, which are iteratively refined during the analysis procedure. These event models are supplied to the individual components (2), where they are used for local analysis (3) that provides the response times of each task mapped to the resource on the basis of the local scheduling analyses. Like the timing of the incoming events, also the timing of the events produced by the tasks can be captured by event models. These are determined using the results of the local response time analysis. The common assumption is that tasks produce exactly one output event for each activating event. The distance between events at the output of the task is then mainly a function of the distance between events at the input and the task's response time jitter.

The resulting output event models can in turn be input event models to other components, or outputs to the environment. During the analysis iteration, the output event models are compared to those used in the previous analysis iteration (4). If all are the same, the analysis has converged, otherwise the corresponding local analyses are repeated with the refined inputs.

For systems containing feedback between two or more components, initial event models are required to begin the local analysis. A solution for this problem is the *starting*

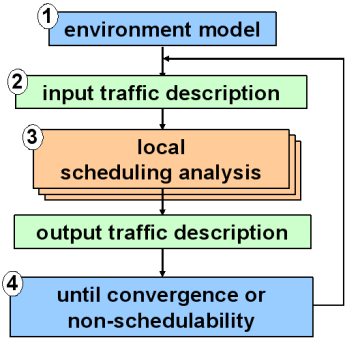

Figure 2.1: Compositional System Level Performance Analysis Loop.

point generation as proposed in [Ric03]. The starting-point is generated by duplicating the environmental input event models along all task chains until an initial activating event model is available for each task. This starting point is possibly optimistic. After a local analysis, output event models can only become more generic, and thus subsume the previous estimates. This monotonic behavior ensures that a fixed point is eventually reached. If the process does not converge, the event model estimates may grow until some timing constraint is violated, and the system cannot be deemed schedulable.

The suggested fixed-point iteration raised concerns about the convergence of the procedure and the correctness of results. A stringent, but only partially formal argumentation is included in [Ric02b] and [Sch09b]. The validity of the approach was shown more formally in [Jon08, Liu08, Ste11] — but only with respect to the constrained system model in which task activating event models are the only analysis parameter. We build on this analysis concept to tackle the analysis dependencies of more complex multiprocessor system setups with shared resources.

The remainder of this chapter first formalizes our notion of an analyzable system model and decomposition. Then we provide a concrete set of parameters which we intend to investigate, and finally specify the procedure to integrate the analysis results. We will then also identify the set of conditions with which the individual analyses have to comply in order to allow a successful integration.

2.2 Modeling of Real-Time Systems

To facilitate reasoning about the run-time properties of an actual system, the system behavior has to be represented in an abstract model. Because an actual system will exhibit an intractable amount of possible states, such a model has to rely on

abstraction and simplification. The purpose of the model is to capture the known and relevant details of the system, so that an analysis can be applied that extrapolates further details that are of interest but not obvious from the system specification.

2.2.1 System Model

Let the actual real-time *system* under investigation exhibit a set of *system properties* that we want to investigate. For the purpose of analysis, our goal is to establish a *system model* that allows to predict these system properties. The model consists of a set of *model parameters*. We assume that we can predict specific system properties that we are interested in from a set of model parameters with the help of a *prediction function*. It is no loss of generality to assume that each property can be provided by a dedicated prediction function. We say that a model *represents* a system if it allows predictions about all properties of interest.

The relationship between the system properties, model parameters, and prediction function is illustrated in Figure 2.2. As an example, a system property of interest may be the maximum end-to-end latency of an application. The corresponding model parameter is than simply a scalar and the corresponding prediction function interprets the scalar as an upper bound on the actual latency. Another system property may be the maximum number of events at the input of a task. This could be captured through an event model as the corresponding parameter.

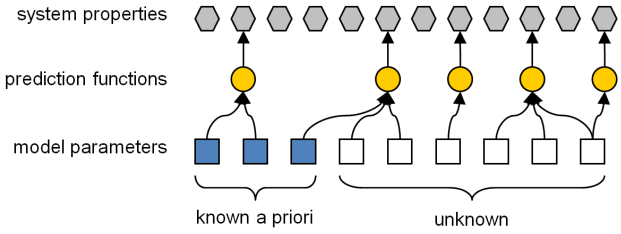

Figure 2.2: System Model

The result of a prediction function must of course be *comparable* to the system property that it predicts. A system property is *conservatively predicted*, if the actual value of the property is contained in the bound provided by the corresponding prediction function. With this in mind, we can define conservativity indirectly also for model parameters.

Definition 2.1 (Conservativity of a Set of Model Parameters)**.** *A set of model parameters is said to be* conservative *if every prediction function that has these (and possibly other) parameters as an input conservatively predicts the corresponding system property.*

Although this will not be exploited in the present thesis, this definition allows for the presence of parameters for which one can not determine conservativity in isolation, but which rather are valid only in relation to other parameters. As an example, task activation offsets are often used to augment the response time analysis (as in e.g. [Hen06b]). But an offset alone can not be deemed "conservative" without considering the event models to which it refers. Similarly, there may be a correlation between two event models, for example at the output of a multiplexer. Then each event is either in the one stream or the other. Assuming conservativity for each individual event model implies a large overestimation with respect to the total load. Instead, it is in many (but not necessarily all) respects more accurate to reason about the tandem of the parameters.

We have a *conservative system model*, if all parameters of the model are conservative. Deriving such a system model is the key goal of our analysis. Because the system model is an abstraction of the actual system behavior, it typically does not cover every detail of the system, but settles at a convenient abstraction level that provides a reasonable trade-off between speed and accuracy (see [Sch08a, Per08] and [Wig09] for an interesting discussion on what level of detail can be achieved with which abstraction).

For the scope of performance analysis, we are interested only in modeling those properties of a system that have a direct or indirect impact on the *timing*, mainly the bounds on the times at which specific states may be observed. We denote the relevant model parameters as *timing parameters*. It is not uncommon that the timing is influenced by functional aspects. For example, a task's worst-case execution time can be influenced by the value of the data to be processed. In this case, the value of the data can be seen as a timing parameter. Quite often an accurate modeling of such value-dependent behavior is waived for the sake of a compact model (e.g. in [Lee87, LeB01, Cha03a, Ric03]). We will introduce the model parameters relevant in our type of systems in Section 2.4.

The model parameters have a predefined relationship that is implied by the behavior of the actual system: for example interesting parameters such as the availability of data in the system, the time of data consumption and production of the tasks, and the arbitration of the available resources all depend on the timing of incoming data. These types of dependencies allow a sufficiently specified model to make predictions about the previously unknown parameters, as shown in the following sections.

2.2.2 Analysis Baseline

In order to perform our analysis, we assume that a sufficiently large subset of the model parameters is *known a priori*, i.e. that these parameters have conservative values that are specified by the designer. These parameters are then the baseline to establish the remaining parameter's values. For example, the configuration of the time-triggered task activations, or the pattern of external events that the system is subjected to are typically known a priori. By contrast, there is usually no initial

knowledge about the run-time dependent parameters "inside" the system, such as the task's output event models or cache state.

Most generally, we expect the system model to have a well defined set of parameters and a *global analysis function* that relates the model parameters to each other. This global analysis function takes the a priori known parameters as an input, and returns all unknown parameters as a result. This is captured in the following definition.

Definition 2.2 (System Model)**.** *A system model y consists of*

- *a* parameterization PS_y*, where each parameter value* v_p *is from its own domain* $\forall p \in PS_y : v_p \in D_p$*.*
- *a* set of parameters $KPS_y \subseteq PS_y$ *for which the (conservative) valuations are known a priori.*
- *a* global analysis function $F_y : \prod_{p \in KPS_y} D_p \mapsto \prod_{p \in PS_y} D_p$ *that computes the values of all parameters in* PS_y *from the set of known parameters.* $\prod$ *is the Cartesian product of the individual domains.*
- *a* set of prediction functions *that allows establishing the actual properties from the model parameters in* PS_y*.*

Obviously, given a certain system complexity, it is not feasible to directly provide such a global analysis function. Therefore, the next section introduces measures to *decompose* the analysis problem.

2.3 Analysis Decomposition

To break down the analysis complexity, this section introduces the notion of analysis functions that each are only concerned with a subset of parameters of the system model.

The benefit of this procedure is illustrated in Figure 2.3. Instead of supplying one global, monolithic analysis function that directly derives all unknown parameter values, we rely on a *set of analysis functions*, each of which is only concerned with a subset of the unknown parameters of the system model. This decomposition is the main idea behind the compositional performance analysis: For each parameter, a dedicated analysis can be provided that specializes on the relevant effects. Then, if the system (model) changes, only the affected analysis functions have to be adapted (see also [Ric04]).

Analysis functions capture the dependencies between the system parameters (for example the timing relationship between the events at the input of a task and its outputs). Thus, each analysis function relies on a subset of the (known) analysis parameters as an input, and some internal properties of the investigated component. The *analysis functions* are defined as follows:

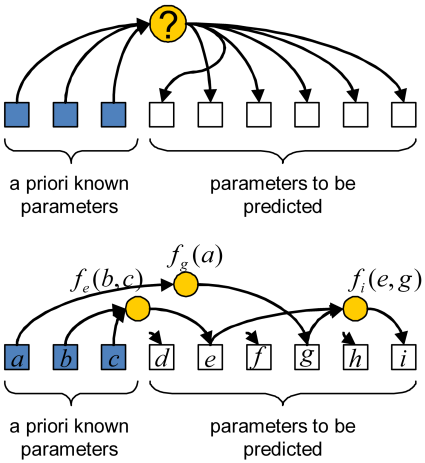

Figure 2.3: Decomposition

Definition 2.3 (Analysis Function)**.** *An* analysis function f_p *computes the value of the system parameter* $p \in PS$ *from a set of input parameters* $IPS_p \subseteq PS$*:*

$$f_p : D_{f_p} \mapsto C_{f_p} \tag{2.1}$$

where

- PS *is the the set of model parameters that represents the system*
- C_{f_p} *is the domain of the analysis result, i.e. that of the computed system parameter:* $C_{f_p} = D_p$
- D_{f_p} *is the Cartesian product of the domains of the input parameters* $\{i_1; i_2; \ldots; i_n\}$ *of* f_p*:*

$$D_{f_p} = \prod_{i \in IPS_p} D_i = D_{i_1} \times D_{i_2} \times \ldots \times D_{i_n} \tag{2.2}$$

In addition to the input parameter set D_{f_p} that is a subset of the model parameters, the analysis function can also consider parameters implicitly (for example the scheduling parameters such as task priorities, or the number of tokens in a cycle). As long as these parameters are known and constant (not dependant on other model parameters) there is no need to make them an explicit part of the system model.

Such an analysis function will also be called a *local analysis*, because it does not require a global view on all system parameters. We will see in Section 2.4.1 that the

global analysis function defined in Definition 2.2 is not simply the collection of local analysis functions due to dependencies between the parameters.

We have defined that each analysis function can have multiple input parameters but only one model parameter as a result. This is no loss of generality, because analyses that compute sets of parameters can be modeled by a set of analysis functions. But as noted above (after Definition 2.1), there may be sets of parameters that allow adequately expressing a system property only in tandem (for example to express correlations between different event models at the output of a multiplexer). Such parameter sets can make the identification of conservativity and the definition of measures for comparison difficult: For example, the offset between task activations has a non-monotonic influence on the tasks response times [Jer04, Rac08]. An explicit treatment of such parameter sets would overload the argumentation in this chapter and it is not required by the analyses presented in this thesis. However, in our model the complexity of each parameter is irrelevant, and each parameter may actually consist of multiple dimensions that represent different aspects. Thus, if inter-parameter correlations are required for an accurate analysis, an intermediate "multi-dimensional parameter" can be defined. It is then up to the details of the analysis function to interpret and compute these complex parameters in tandem. The conditions identified in the remainder of this chapter (such as *comparability*) must hold for the multi-dimensional parameters, but not necessarily for each of their internal dimensions.

We demand that all analysis functions are *correct*, according to the following definition:

Definition 2.4 (Correctness)**.** *An analysis function is* correct*, if the computed parameter p is conservative under the assumption that its input parameters are conservative.*

The notion of correctness is introduced (as opposed to the stronger demand for conservativity), because the iterative procedure that we present in Section 2.5.2 initially invokes analysis functions with input parameters that may not conservatively reflect the actual system behavior. An analysis can then not be expected to deliver conservative results. Correctness does however imply that if the input parameters are conservative, then so is the analysis result.

Note that the property of *correctness* is maintained when two correct analyses are composed. This is trivially true if the analysis do not share parameters. If the analysis share parameters (i.e. the computed model parameter of one analysis is an input of the other), then the conservativity assumption in Definition 2.4 holds transitively.

Sufficient Specification of the Model

The decomposition of the global analysis function raises the question of sufficient specification of the model. It is only possible to establish the value of a model parameter if the corresponding analysis function has all its input parameters. The

model parameters may even transitively and cyclically depend on each other. This can lead to problematic under-determination, in which one can not decide which solution is conservative (except usually for the trivial infinite bound). The problem can be illustrated with an actual application topology. Consider a system which contains two event-driven tasks in a simple cycle without external inputs. At run-time, these tasks will execute with an varying rate that depends on their execution times. Now, let the system parameterization contain only the task activating event models of each task (this is the modeling assumption in e.g. [Ric02a]). All of these event models are initially unknown. Due to the cyclic dependency, no bound can be established about the actual activation jitter or average activation distance of either task. Thus, in order to determine the behavior of the cycle additional assumptions or parameters are required, such as a bounded number of tokens in the cycle or an external input that limits the number of activations of one task (as in [Jer05]). The corresponding analysis function can than conservatively determine its output parameter (the respective task's output event model) even when the other cycle-internal event models are indeterminate. Based on this, the reasoning about the timing of all other tasks in the cycle can follow.

The following definition provides a general condition for the analyzability of a system parameterization including analysis functions with cyclic dependencies.

Definition 2.5. *Let $G_D(Y)$ be the* dependency graph *of a system model Y. The vertexes in this graph represent the model parameters. An directed edge exists from every parameter p to all parameters q, for which the respective analysis functions f_q have model parameter p as an input.*

A system model is sufficiently specified *when*

- *every parameter of the system that is of interest is either known a priori or all input parameters of the corresponding analysis function are in $G_D(Y)$; and*
- *in every cycle $g_c \subseteq G_D(Y)$ there is at least one a priori known parameter* or *at least one analysis function that provides a conservative output under the condition that all parameters that are not part of g_c, but on which analyses in g_c rely, are conservative.*

As illustrated above, the demand for a sufficiently specified system is reasonable. In acyclic dependency graphs, there must of course be an a priori known parameter at the beginning of each chain. In the presence of cyclic dependencies, the second demand addresses the problem of an under-determined set of dependent functions. It will be shown in Theorem 2.1 that this specification is sufficient to establish conservative parameters throughout the model.

2.4 A Self-Contained Set of Analysis Functions For Multiprocessor Systems

In order to apply our decomposition concept the actual multiprocessor real-time systems presented in Section 1.6.1, we provide specific parameters and corresponding analysis functions. This system model is the baseline for the analysis provided in the successive chapters. The reasoning in the present chapter is however independent of the actual parameters.

Analysis Scope

As motivated in Section 1.4.1, we aim to investigate multiprocessor systems that consists of multiple dynamically scheduled tasks that can trigger each other and have access to shared resources during their execution. Thus, tasks can be dependant both via chaining and the use of shared resources.

Let the system consist of hardware and software components: the hardware platform is given by a set of *resources*, which can be processors, buses, or memories. The software is given by a set of *tasks* that together implement a certain application. A *task* is a time-consuming procedure that requires a bounded execution time on the resource that it is mapped to. A task can implement a computation, a communication, or a storage operation. Every task is *mapped* to a specific resource. The *resource* assigns execution time to the tasks according to a run-time or off-line *scheduling policy*. During execution, a task may request services from a secondary *shared resource*. A *shared resource operation* consists of a *shared resource request*, a *shared resource service* provided by the shared resource, and a *shared resource response*. The shared resource dynamically *arbitrates* between conflicting requests from different sources. The difference between the number of issued requests and received responses is the number of *open* shared resource operations of a task.

System parameterization

We will now introduce a specific parameterization that allows to represent the behavior of such systems. The main purpose is to provide a self-contained set of parameters and analysis functions that enables the derivation of internal, a priori unknown parameters.

At this point, each model parameter and analysis function is defined in an abstract fashion, because different implementations are conceivable. These will be the concern of the successive chapters. Static attributes of the system, such as hardware topology, the task-to-processor mapping, the task execution times, assigned scheduling policies and its parameters are considered as *implicit* parts of the model. During system analysis, these parameters are constant and *known* and thus need not be the output of an analysis function. Formalizations of these properties will be introduced in the respective chapters as needed.

Definition 2.6 (MPSoC System parameterization)**.** *The parameterization of a multiprocessor system with shared resources is defined as follows:*

$$PS = PS_{TAE} \cup PS_{SRRB} \cup PS_{SRDB} \cup PS_{LTB} \cup PS_{LAT} \tag{2.3}$$

where

- PS_{TAE} *is the set of parameters representing for each task the* task activating event model*:* $\forall p \in PS_{TAE} : D_p = D_{TAE}$*.*
- PS_{SRRB} *is the set of parameters representing for each task the* shared resource request bound*:* $\forall p \in PS_{SRRB} : D_p = D_{SRRB}$*.*
- PS_{SRDB} *is the set of parameters representing for each task and shared resource the* shared resource delay bound *that the task may experience:* $\forall p \in PS_{SRDB} : D_p = D_{SRDB}$*.*
- PS_{LTB} *is the set of parameters representing for each task the* local timing behavior*:* $\forall p \in PS_{LTB} : D_p = D_{LTB}$*.*
- PS_{LAT} *is the set of parameters representing for each path in the system the* end-to-end behavior*:* $\forall p \in PS_{LAT} : D_p = D_{LAT}$*.*

The Task's Timing Behavior (LTB) The major concern in a real-time system is the timing of the task executions, and whether their execution complies with the constraints imposed on the system. The abstract metric of the *local task timing behavior, LTB* of a task captures the timing of its possible activations and completions, including any delaying effects due to scheduling or resource conflicts.

A basic implementation of this parameter is the task's worst-case response time (WCRT, as used in [Ric02b]). In this case the domain of the D_{LTB} is a scalar. In network calculus and real-time calculus [LeB01, Thi00, Cha03a] the abstraction is provided by a combination of the event arrival and resource service curves, which are continuous functions. A expressive yet efficient discrete metric called the *multiple event busy time* has been suggested in [Sch08c]; this metric will be the focus of Section 3.5 and Chapter 6.

We can already say that the local timing behavior of a task τ_i at least a function of the activating event models of all tasks sharing the processor (modeled by parameters PS_{TAE}) and their shared resource delay bounds (parameters PS_{SRDB}):

$$f_i^{LTB} : D_{TAE}^* \times D_{SRDB}^* \mapsto D_{LTB} \tag{2.4}$$

The Task-Activating Event Models (TAE) The derivation of the task-activating event models was the key concern of previous compositional performance analyses [Gre93b, Ric02b, Cha03a]. We review the respective models in Chapter 3 and introduce a new and generic method to derive the task's output event model from the task's activating event model (which is a parameter from PS_{TAE}) and the task's

local timing behavior (from PS_{LTB}). The output event model analysis function f_i^{TAE} of a task τ_i thus can be defined as follows:

$$f_i^{TAE} : D_{TAE} \times D_{LTB} \mapsto D_{TAE} \tag{2.5}$$

The End-To-End Latency (LAT) Based on the timing of the individual tasks, it is also possible to reason about the timing of task chains. As we will see in Chapter 4, the accuracy of the end-to-end estimate highly depends on the expressiveness of the model of the task timing for each involved task (modeled by parameters PS_{LTB}). Also, the pattern of events (parameters PS_{TAE}) entering a path impacts the run-time unfolding of events, and thus their latency. Generally, the latency of a path P is at least a function with the following parameters:

$$f_P^{LAT} : D_{TAE}^* \times D_{LTB}^* \mapsto D_{LAT} \tag{2.6}$$

The Load Imposed on the Shared Resources (SRRB) To be able to compute the possible interference on a shared resource (as motivated in Section 1.4.1), models on the maximum amount of requests issued by each task are required. These per-task bounds can be subsequently combined to cover all tasks on a processor. The *shared resource request bound (SRRB)* parameters, which are the focus of Chapter 5, describe the possible timing of the shared resource operations initiated by each task. As such, it is related to the task-activating event model description, and can be specified by the minimum and maximum number of operations per time interval. More accurate models may contain additional information such as for example the involved memory addresses, conditional event occurrence, or a distinction between read and write operations. As these parameters impact the actual timing, considering the details can lead to improved analysis results. Commonly however, the analysis improvement comes with the associated cost of a larger analysis time.

The corresponding analysis $f_{i\to S}^{SRRB}$ provides this bound for a task τ_i and shared resource S. For this, we need to consider the task's activating event model, the event models of possibly interfering tasks (parameters PS_{TAE}), and a description of the tasks' shared resource operations (which is constant during the analysis and assumed to be implicit in the analysis function).

$$f_{i\to S}^{SRRB} : D_{TAE}^* \mapsto D_{SRRB} \tag{2.7}$$

The Shared Resource Access Delay (SRDB) Depending on the applied resource arbitration policy, coinciding shared resource requests are serviced in a (possibly run-time) deterministic order. The *Shared Resource Delay Bound (SRDB)* metric models these delays. An analysis for several common arbitration policies is the topic of Chapter 7, which relies on the particularly important *aggregate busy time* that represents the total time for which at least one of the task's requests is not finished.

The possible timing behavior of task τ_i's requests to the shared resource s with respect to the shared resource request bounds of all tasks that request a service from s can be computed from with the following function:

$$f_{i\to s}^{SRDB} : D_{SRRB}^{*} \mapsto D_{SRDB} \tag{2.8}$$

2.4.1 Dependencies between Analysis Functions

According to our definition of the system model (Definition 2.2), each parameter in PS has to be associated with an analysis function. This function has the respective parameter as a result and, as sketched above, requires a subset of the other parameters as its input. Such a set of dedicated analysis functions leaves us with various dependencies that reflect the dependencies of the behavior in the actual system.

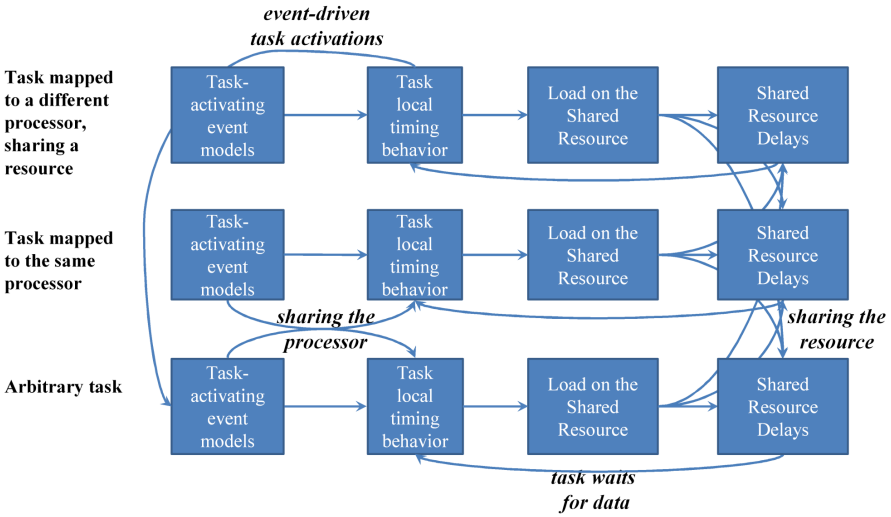

Figure 2.4: Analysis Dependencies in the Presence of Shared Resources (not complete).

Recall the example in Figure 1.5, where three tasks execute on two processing cores with a shared resource. We have highlighted in Section 1.4.1, that there is a mutual dependency between the response times of the tasks on both cores in the physical system. This dependency is now echoed in dependencies between the decomposed analysis functions. This is illustrated in Figure 2.4. The nodes of the graph represent the parameters of the system model, and the edges represent the analysis functions,

the direction from their input parameter to the computed results (some transitive dependencies are not shown).

Acyclic dependencies in this graph are not challenging to resolve. A legal sequence of analyses can then be applied, in which every analysis function is provided with the required input parameters. In general however, there can be various cyclic dependencies that hinder a straight-forward analysis procedure.

2.5 Analysis Composition

In this section, we investigate how the local analyses can be composed into a global analysis function, and how the analysis dependencies can be tackled with the help of an iterative approach.

2.5.1 Finding a Conservative Set of Parameter Values

According to the system model in Definition 2.2, each parameter in the system can be computed with a specific analysis function. Each analysis function as provided in Definition 2.3 only has a "local" scope, as it considers only a subset of the parameters in the system in its computation, and computes only a single parameter (note that this is only a modeling assumption, in a practical implementation a "local analysis" can of course compute more than one parameter at once for efficiency).

As stated before, the analyses are partly dependent on each other as seen in Figures 2.3 and 2.4. Thus the analysis functions form an entangled "net" of equations via the parameters that are the result of one analysis and the inputs of another. Additionally, a subset of the parameters are known a priori, in which case these are never the result, but only inputs to some analyses.

The challenge is now to find a solution to this equation system. We call a set of parameter values that solves the equations "consistent". Computing an analysis function on the base of the consistent parameter values will not result in an analysis result that contradicts the parameter values assumed by any other analysis. Formally, we define:

Definition 2.7 (Consistent Set of Parameter Values)**.** *A set* $\mathbf{v}$ *of parameter values is consistent, if*

$$\forall p \in PS_y : f_p(\pi_{f_p}(\mathbf{v})) \leq \mathbf{v}_p \tag{2.9}$$

where

- $\pi_{f_p}(\mathbf{v}) : \prod_{q \in PS_y} D_q \mapsto D_p$ *is the projection of the consistent parameter values to the actual input parameters required by analysis function* f_p *and*
- $\mathbf{v}_p$ *is entry in* $\mathbf{v}$ *that holds the value of* p*.*

If such a consistent set of parameters is known, all analysis results are also consistent with the provided a priori parameters. Because these are conservative, also each analysis result must be conservative. This is provided in the following theorem.

Theorem 2.1 (Conservativity of Analysis Result)**.** *Assume a sufficiently specified system model and a consistent set of parameter values. Then, each analysis function delivers* conservative *results, if*

1. *all a priori known parameters have conservative values;*
2. *for every a priori unknown parameter, there is a* correct *analysis function that has this parameter as a result.*

Proof. Let the graph $G_D(Y)$ represent the dependencies between the analysis functions, such that each vertex is a model parameter (which is either known a priori or associated with an analysis function), and a directed edge leads from each parameter p to all parameters q, whose analysis functions f_q have require p as an input. The graph $G_D(Y)$ can be cyclic and consist of multiple partitions.

The proof is in two parts: First, we show that all parameters in a cycle are conservative if the external inputs are conservative. Then we show that all external inputs must be conservative.

Part I: All parameters in a cycle are conservative The proof is by contradiction: Assume the result of at least one analysis function f_p is not conservative. Because the analysis result of f_p is consistent with its input parameters, there are only two possibilities in order for p to be non-conservative: Either f_p is not correct. This contradicts assumption (2.). Or, at least one of f_p's input parameters is not conservative. This can be ruled out as follows: Let $C(p)$ be the cycle of which p is a part (which might also consist only of p). Because the model is sufficiently specified, at least one parameter in $C(p)$ is known a priori or an analysis function in $C(p)$ provides conservative results if the parameters along all cycle-external incoming edges are conservative. Let us for now assume that this is the case and let the conservative parameter be c. There are two cases: *case a.)* If the non-conservative input parameter of f_p is c then we have a contradiction either to assumption (1.) or to the assumption that the system is sufficiently specified. *case b.)* If the non-conservative input parameter of f_p is the result of another analysis function, the argument repeats. Because in the cycle all analysis functions are connected, this must ultimately lead to a contradiction. Thus, if all external inputs are conservative and all parameters in the cycle are consistent, then all parameters in the cycle are conservative. It now remains to show that the external inputs to $C(p)$ must be conservative so that we again have a contradiction.

Part II: All parameters in the acyclic dependency graph are conservative. Let G_{DAG} be the condensation of $G_D(Y)$, such that all cycles in $G_D(Y)$ are collapsed into single vertexes, which unify all external edges to and from the respective cycle. Then from the above reasoning, we know that all outgoing edges hold conservative parameters if all incoming edges are conservative. We now show (again by contradiction) that every parameter in G_{DAG} must be conservative. Assume that a parameter q in G_{DAG} is not conservative. Because its analysis function f_q is correct, and q is consistent

with the input parameters of f_q, at least one of f_q's input parameters must be non-conservative. If the non-conservative input parameter is known a priori, we have a contradiction to (1.). If the non-conservative input parameter is the result of an analysis function, the argument repeats. Transitively, this must always lead to a contradiction. □

Note that for systems with cyclic parameter dependencies, Theorem 2.1 relies on the assumption that at least one parameter in every cycle is known to be conservative. We have already given an argumentation based on [Jer05] how this can be achieved for functional cycles when defining sufficient specification in Definition 2.5. For non-functional cyclic dependencies, such a property can be more difficult to establish. However, as shown in [Sch05, Jon08], it will be fulfilled for consistent solutions that are derived by fixed-point iteration. Because, this is exactly the procedure that will be suggested below to find the consistent solution, we can expect this property to be fulfilled also without additional measures (see also [Ste11]).

Hence, the problem of finding a *conservative* set of parameter values can be transformed to the problem of finding a *consistent* set of parameter values for a connected set of analysis functions. A consistent set of parameter values is free of contradictions. One such set of parameter values is given by the case, where the result of every analysis function is *exactly* the value assumed by all other analysis functions. Given such a consistent set of parameter values $\mathbf{v}$, recomputing the result of any analysis function results in the exact same value given in the parameter set itself:

$$\forall p \in PS_y : f_p(\pi_{f_p}(\mathbf{v})) = \mathbf{v}_p \tag{2.10}$$

Thus, every *fixed-point* of the analysis functions is also a consistent set of parameter values. With this in mind, the problem of finding a set of consistent parameter values can be solved by finding a fixed-point.

The following section provides an iterative analysis approach to find such a fixed-point of parameter values. First, we define a global analysis procedure in order to apply the available fixed-point theory. This theory allows us to reason about conditions for the existence of a fixed-point and provides indications on how to find it. Afterwards, we will break down the conditions for the global analysis function back to conditions for the local analyses.

2.5.2 Finding a Consistent Set of Parameter Values

In section 2.4 we have introduced several analysis functions that allow deriving specific system parameters from a subset of all parameters in the system. To rely on the results of fixed-point theory, these individual analysis functions will in this section be combined to constitute a *global analysis function* f_{gaf}. The idea is that a consistent

solution for the global analysis function will also be a consistent solution for the individual functions.

Due to the analysis inter-dependencies a consistent set of parameter values can not be derived in a single iteration of the global analysis function. Initially, only a subset of the parameters are actually known: Internal parameters such as the delays for the shared resource accesses can not be conservatively specified a priori. To still allow for computing analysis results, an initial assumption has to be made for all parameters that are a priori unknown. These assumptions will be captured in the *intermediate analysis state*, that will be refined by the analysis functions.

Definition 2.8 (Intermediate Analysis State)**.** *An* intermediate analysis state *is a valuation of the parameters in the model's parameter set PS. It is represented as a vector* $\mathbf{i}$*, the entries of which are the values of a specific parameter.*

$$\mathbf{i} \in I, I = \prod_{p \in PS} D_p \tag{2.11}$$

We will later return to the problem of constructing an initial analysis state ("starting point" as in [Ric03, Ste08, Ste11]). But once we have specified such a state, we can perform any local analysis. The following definitions specify this procedure, but purposely leave open the question of which analysis or sequence of analyses is actually chosen. We will later see that under the conditions provided, any sequence of analyses will lead to a fixed-point (actually, the least fixed-point).

According to Definition 2.3, each analysis function produces exactly one parameter value as a result, and thus each entry in the vector $\mathbf{i}$ can be associated with exactly one analysis function. The idea is to "store" in this vector the most up to date analysis results: Whenever an analysis has been performed, the corresponding element in the vector is updated with the new result. A fixed-point has then been found, when performing any analysis function again results in the same vector.

The *global analysis function* is defined hierarchically: First, a *subset analysis* is introduced which simply performs a sequence of local analyses (specified in the set σ) based on the current intermediate analysis state $\mathbf{i}$. A priori known parameters ($p \in KPS$) which are not the output of an analysis are not affected. The subset analysis returns an updated intermediate analysis state based on the results of the individual analysis functions.

Definition 2.9 (Subset Analysis)**.** *A Subset Analysis A consists of recomputing the parameter values $\sigma \subseteq PS$ based on the parameter values specified in the intermediate analysis state* $\mathbf{i}$*:*

$$A : \mathcal{P}(PS) \times I \mapsto I \tag{2.12}$$

$$A(\sigma, \mathbf{i}) = (..., \begin{cases} f_p(\pi_{f_p}(\mathbf{i})), & \text{if } p \in \sigma \wedge p \notin KPS \\ i_p, & \text{otherwise} \end{cases}, ...)^T \tag{2.13}$$

where

- *PS is the set of investigated parameters in the system. $\mathcal{P}(PS)$ is its power set. Each system parameter is computed by a function f_p according to Definition 2.3.*
- *KPS is the set of a priori known parameters.*
- *I is the vector representation of the intermediate analysis state according to Definition 2.8. i_p is the entry in $\mathbf{i}$ that holds the value of parameter p.*
- *σ is the set of parameters that will be recomputed.*
- *D_f is the domain (input parameters) of the analysis function f.*
- *$\pi_f(\mathbf{i}) : I \mapsto D_f$ is the projection of the intermediate analysis state to the input parameters required by analysis function f.*

Based on the subset analysis, the global analysis function is defined as follows. It begins with the set of all parameter values (PS) and recomputes their values based on the current intermediate analysis state $\mathbf{i}$. For this, it repeatedly selects a subset of the parameter values in function $\sigma(PS)$. An iteration of the global analysis function is complete if every analysis of which the input parameters have changed since the last iteration has been recomputed at least once. We then also call such an analysis and its computed parameter *up-to-date*.

Definition 2.10 (Global Analysis Function)**.** *The global analysis function f_{gaf} is defined as follows*

$$f_{gaf} : I \mapsto I \tag{2.14}$$

$$f_{gaf}(\mathbf{i}) = f'(\mathbf{i}, PS) \tag{2.15}$$

$$f'(\mathbf{i}, P) = \begin{cases} f'\left(A(\sigma(PS), \mathbf{i}), \{P \setminus \sigma(PS)\}\right) & \text{if } P \cap P^{\neq} \neq \emptyset \\ \mathbf{i} & \text{if } P \cap P^{\neq} = \emptyset \end{cases} \tag{2.16}$$

where

- *PS is the set of modeled parameters in the system*
- *$\sigma : \mathcal{P}(PS) \mapsto \mathcal{P}(PS)$ selects a non-empty subset from the set of the parameters to be analyzed in each recursion of (2.16).*
- *$P^{\neq} \in \mathcal{P}(P)$ is the set of system parameters (analysis results) for which the input parameters have changed since the invocation of f_{gaf}, i.e. which are initially not up-to-date.*

In the definition, the function $f_{gaf}(\mathbf{i})$ redirects to the function $f'(\mathbf{i}, P)$ with the set of parameters to be reevaluated. The recursive call of f' continues until every parameter in PS was analysed at least once. Thus, if $f_{gaf}(\mathbf{i}) = \mathbf{i}$, we can be sure that $\mathbf{i}$ is a fixed point. Note that an actual implementation does not have to be aware of the set of "not-yet analysed parameters" $P^{\neq}$, but we consider an iteration of the global analysis function as finished only if this set is empty. Thus, just as any deterministic

scheme is allowed, a random selection of analysis functions to be reevaluated would also be allowed and within the scope of Definition 2.10.

Finally, if we have a fixed-point of the global analysis function ($f_{gaf}(\mathbf{i}) = \mathbf{i}$), this implies that we have $\forall p \in P_y : f_p(\Pi_{f_p}(\mathbf{i})) = \mathbf{i}_p$. Thus, the converged "intermediate" analysis state represents also a consistent set of parameter values that we are looking for for the entangled set of local analysis functions. Thus, this fixed-point is conservative:

Lemma 2.2. *Every fixed-point of the global analysis function of Definition 2.9 is conservative.*

Proof. Theorem 2.1 has established that any consistent set of parameter valuations is conservative. Every fixed point found by Definition 2.10 is also a consistent solution (because it fulfills equation (2.10)). Thus, every such fixed-point is conservative. □

2.5.3 Solving Analysis Dependencies with Fixed-Point Theory

We have now defined a global analysis function as an arbitrary composition of local analysis functions. Furthermore, we have seen that if we have a fixed-point of the global analysis function, then we have conservative model parameters. It now remains to find a procedure to derive such a fixed-point. In this section, fixed-point theory is briefly revisited to find the necessary preconditions for the application of an iterative analysis approach. A key observation in the successive section is that these conditions can be broken down to conditions for the individual analysis functions.

The concept of using fixed-point theory for the analysis of event-driven multiprocessor systems has already been applied in [Ric02b, Sch05, Jon08, Liu08, Ste08], where it was used to tackle non-functional cyclic dependencies in (mainly) multi-node ("distributed") systems. Also there, each analysis iteration consists of a concatenation of local (response-time) analyses, which map the current intermediate analysis state (i.e. the task activating event models) to an updated analysis state (i.e. the task output event models). The monotonicity of this concatenation is obviously ensured if each involved local analysis is monotonic. Given that tasks do not share resources, this will be the case due to the nature of the classical worst-case response time analysis, the result of which can only increase when subjected to more generic event models [Ste11]. Therefore, the condition for the existence of a fixed-point is fulfilled for the traditional analysis. The presence of shared resources and other investigated system parameters however challenges this assumption.

We will now revisit key theorems from fixed-point theory to identify the necessary conditions for our analysis procedure to find a fixed-point. Firstly, we can rely on the following lemma to establish that a fixed-point exists.

Lemma 2.3 (Existence of Fixed-Point, from [Tar55])**.** *Any order preserving function defined on a complete partially ordered set (CPO) has itself at least one fixed-point.*

This lemma is refined in the following theorem, which additionally gives an indication on how the least fixed-point can be found.

Theorem 2.4 (Kleene Iteration, from [Dav90, Bou49, Wit51, Ste11])**.** *An order preserving function function f defined on a complete partially ordered set has a least fixed-point $\dot{\mathbf{v}}$ that can be found by the Kleene iteration:*

$$f^0 \leq \dot{\mathbf{v}} \tag{2.17}$$

$$\Rightarrow \quad \dot{\mathbf{v}} = \sup(\{f^n(\mathbf{v}^0), n \in \mathbb{N}\}) \tag{2.18}$$

where $f^n(\cdot)$ is the n-fold recursive application of the function, and $\mathbf{v}^0$ is an initial value that is smaller than the fixed-point.

In addition to Kleene's original reasoning, it is shown in [Ste11] that the initial starting point $\mathbf{v}^0$ does not necessarily have to be the least element of the CPO ($\perp$), but can be any element of the CPO that is smaller than the smallest fixed-point. In practice this can lead to a reduced number of iterations. Also, with respect to conservativity, it is there shown that the fixed-point that is reached when starting with the least element is also conservative (in addition to [Jon08] where the starting point was demanded to be a "simulatable trace"). Based on Theorem 2.4, we can define the following iterative system analysis procedure to emulate the Kleene iteration.

Definition 2.11 (System Analysis Procedure)**.** *The system analysis procedure of a system with parameter set PS and a set of a priori known parameters $KPS \subseteq PS$ is defined as follows:*

$$F_{sap} : \prod_{p \in KPS} D_p \mapsto \prod_{p \in PS} D_p \tag{2.19}$$

$$F_{sap}(\mathbf{v}_k) = \min\{\mathbf{i}^n \mid \mathbf{i}^n = \mathbf{i}^{n+1}\} \tag{2.20}$$

$$\mathbf{i}^0 = (v_1, v_2, \ldots, v_n)^T \circ (\perp_{p_1}, \perp_{p_2}, \ldots, \perp_{p_m})^T = \perp_{PS}, \tag{2.21}$$

$$\mathbf{i}^{n+1} = f_{gaf}(\mathbf{i}^{n+1}) \tag{2.22}$$

where

- *f_{gaf} is a single iteration of the global analysis function in Definition 2.10;*
- *v_1 to v_n are the values of the a priori known parameters;*
- *p_1 to p_m are the a priori unknown parameters; and $\perp_p$ is the least element of its respective domain D_p.*

2.5.4 Conditions for Analysis Functions

Now, above we defined our system analysis on the basis of the *global* analysis function, which in fact is a collection of repeated updates of analyses with *local* scope. We will now break down the convergence conditions stated above for Kleene's iteration to conditions for the individual analysis functions.

From Theorem 2.4, we can deduce the following conditions for the global analysis function f_{gaf} in order for the analysis iteration to find a fixed-point:

Corollary 2.5 (Condition for Convergence of Global Analysis Function)**.** *The recursive application of the global analysis function according to Definition 2.11 converges towards a fixed-point, if*

a) The global analysis function f_{gaf} is monotonic with respect to the intermediate analysis state.

b) The domain of the intermediate analysis states forms a complete partial order.

The monotonicity condition for the global analysis function can be easily decomposed into conditions for the individual functions. This is provided by Lemma 2.6:

Lemma 2.6 (Monotonicity of Global Analysis Function)**.** *The global analysis function (defined in Definition 2.10) is monotonic with respect to the intermediate analysis state, if each analysis function is monotonic with respect to their input parameters:*

$$\forall p \in PS; \mathbf{i}_1, \mathbf{i}_2 \in I : \mathbf{i}_1 \geq \mathbf{i}_2 \Rightarrow f_p(\pi_p(\mathbf{i}_1)) \geq_p f_p(\pi_p(\mathbf{i}_2)) \tag{2.23}$$

Proof. The global analysis function is a concatenation of subset analyses (of Definition 2.9). If each subset analysis is monotonic, then so is the global analysis function. The subset analysis is monotonic, if all analysis functions are monotonic. □

Also, the analysis states of the global analysis function form a complete partial order, if a complete partial order exists for the domain of each system parameter:

Lemma 2.7 (CPO of Analysis States)**.** *The domain of the parameter set $\prod_{p \in PS} D_p$ forms a complete partial order, if the domain of each parameter D_p forms a complete partial order.*

Proof. If the domain of each parameter D_p forms a complete partial order, then we know:

- for every parameter $p \in PS$, a partial order $\leq_p$ exists between the elements of its domain D_p; and
- every parameter $p \in PS$ has a smallest element $\forall v_p \in D_p : \bot_p \leq v_p$.

The domain of the intermediate analysis state is the Cartesian product of the domains of the individual parameters. Then a complete partial order $\langle \bot_{PS}, \leq_{PS} \rangle$ of analysis states is given by the smallest element

$$\bot_{PS} \quad = \quad (\bot_1, \bot_2, \ldots, \bot_n)^T \tag{2.24}$$

and the partial order

$$\leq_{PS}: i_1 \leq i_2 \Rightarrow \forall p \in PS : v_p^1 \leq_p v_p^2. \tag{2.25}$$

□

Hence, the conditions under which the repeated invocation of the *global* analysis function according to definition 2.11 finds a fixed point can be broken down into respective conditions for the *local* analysis functions. The following summarizes these conditions.

Corollary 2.8 (Conditions for individual analysis functions)**.** *The recursive application of the global analysis function according to Definition 2.11 converges towards a fixed-point, if*

a) the domain of each system parameter forms a complete partial order;

b) each analysis function is monotonic with respect to their input parameters.

It is of key importance that all analysis modules comply with these conditions. The remainder of this thesis presents several updated and new analysis functions. For each, we will investigate and ensure its compliance.

2.5.5 Speed of Convergence

Given the monotonicity of the involved analysis functions, all parameter values can only become more generic with each iteration. As Kleene showed, this ultimately leads to a fixed-point, actually the smallest possible fixed-point. However, this fixed-point can be arbitrarily large, thus possibly violating the constraints of the system. When this happens, one can abort the remaining analysis, because due to the monotonic progression of the intermediate analysis state, we know that also the fixed-point will violate the constraints.

Lemma 2.3 gives no indication on the number of analysis steps that are required to reach a fixed-point, which depends on the amount of progress made per iteration. In [Ric02b], the progress per iteration is quantified to be at least the size of a "smallest common divisor" of all relevant parameters. This was elaborated in [Ste08, Ste11]. In general, a system will be constrained and thus the space of possible parameter valuations is bounded in every dimension. As each parameter can only assume discrete multiples of the smallest common divisor, only a finite set of parameter values is possible and the set of investigated parameter valuations is finite. Thus, after a finite number of iterations the analysis must either converge or a constraint must be violated.

This reasoning gives a theoretical upper bound on the number of iterations that can be very large. However, in practical applications the number of iterations is actually reasonably small, as will be demonstrated in Chapter 8.

2.6 Summary

In this chapter we have provided a compositional analysis framework for multi-node multiprocessor systems with shared resources. For this we have revisited the theory behind compositional performance analysis and adapted it: With respect to previous

work, we left the predominant focus on task-activating event models as the investigated model parameter and provided a general argumentation that applies to more diverse parameter sets as considered in this thesis. The requirements for finding a conservative solution were mapped to fixed-point theory, which delivered the key conditions for the convergence of the iterative analysis. These conditions could then be broken down to conditions for the individual analysis functions. This allows for an easy extension of the set of analysis functions and investigated model parameters.

Our application of this analysis framework is the multiprocessor setup, in which event-driven or time-driven tasks may share resources across the processor cores. For this, we have presented a suitable set of analysis parameters for which corresponding analyses will be provided in the next chapters.

3 Timing Analysis with General Load Event Models

3.1 Introduction

One of the most important timing aspects in a real-time system is the timing of events that represent the activation or completion of a task or communication. Very often, the completion of one task will lead to the activation of another, the timing of which is then subject to the run-time behavior experienced by the preceding tasks. Sharing resources among tasks inevitably increases the dynamism of events and can lead to more complex event patterns within the system. A key goal of compositional performance analysis approaches is therefore to accurately derive bounds on the timing of events in the system that are not provided as external inputs. The corresponding event models are the basis for determining the amount of interference that any execution or communication in the system may experience. As sketched in Section 2.3 it is possible to derive the timing of events produced by a task from the timing of events that lead to its activation. This problem will be addressed in this chapter. In particular it

- defines the *generalized load event model* as the class of investigated model parameters, and establishes that this parameter exhibits the relevant properties for use in the multiprocessor analysis of Chapter 2;
- investigates existing abstractions of component timing (such as the task's worst-case response time), and introduces the *multiple event busy time* as a new abstraction that provides a convenient trade-off between complexity and modeling detail;
- provides an *event model propagation function* that allows deriving the timing of task activating events within the system from those specified at the system's inputs.

The remainder of this chapter is structured as follows. Next, we introduce our notion of events and how to capture their timing properties with event models. In Section 3.3 basic properties of the generalized load event model are identified. Then, Section 3.4 surveys existing models to capture the service provided by a resource, which leads to the introduction of a new efficient model in Section 3.5. These resource and event models are combined in Section 3.6 to provide the new generalized method to derive the timing of events in the system. Finally, we conduct some experiments to evaluate the new method in Section 3.7 and provide an intermediate conclusion in Section 3.8.

The provided methods become part of the multiprocessor analysis framework, which is again evaluated as a whole in Chapter 8.

3.2 Modeling the Timing of Events

In this section, we first introduce the notion of *events* that establishes the connection between the behavior of the physical system and the system model. Based on this, we can then reason about event models in order express the timing properties of events in the system.

An event is a logical concept that identifies a specific state transition in the physical system. The system model and analysis reason about the occurrence of these events, and any property that is true for the events according to the model should also hold for the state transitions in the physical world which they represent (which provides the "conservativity" of Definition 2.1).

Definition 3.1 (Event)**.** *An* event *is a marker of a point in time at which "something of interest happens".*

For example, an event may mark the expiration of a timer or the arrival of data at the input of a task. If such an event leads to the activation of a task, we also say that the respective event *activates* the task. Likewise, the event that marks the termination of the task can be said to be *produced* by the task. We also say that an event *occurs* at the point in time that it marks.

The timing of events is usually of interest only in relation to other events. In particular, we are interested in the timing of events of the same *event stream*, which is the collection of all events that mark an identical condition, such as the activation of a specific task.

Events in an event stream have an intrinsic timing relationship determined by the system setup. Uncovering and reasoning about this relationship is of crucial interest to the performance analysis. For this purpose, an *event model* is needed that is a collection of properties that hold for all events in an event stream, or even sets of event streams. For example, an event model may provide inter-event timing information, correlations between event streams (offsets), or also functional details (such as memory address sequences). As usual, however, the more detailed the model, the more complex the respective analysis operations become. Notable event models have been introduced e.g. in [Gre93b, Ric02b, Cha05a].

For the scope of our analysis, we consider only those event models that provide the relative timing between events in an event stream according to the following definition:

Definition 3.2 (Generalized Load Event Model)**.** *A load event model provides for every number of events $n \geq 1$ in an event stream*

- *$\delta^-(n)$, the size of the smallest time window within which n or more events may occur, and*

- $\delta^+(n)$, *the size of the largest time window within which less than* n *events may occur.*

Let the domain of load event models be denoted with $\mathbb{E}$.

The event models provided in the literature cited above comply with this definition. A key contribution of the present chapter is to facilitate the derivation of load event models at the output of tasks from those at its input. Previous work [Ric04] already supports the load event model as an input parameter to a local analysis, but has provided means to compute output event models only for *standard event models*. This and other work will be discussed in Section 3.4. As we will see below, the load event model can also be seen as an inverse of the *arrival curve* defined in network calculus [LeB01].

The event distance functions used in Definition 3.2 are defined as follows:

Definition 3.3 (Event Distance Functions)**.** *A minimum event distance function is defined by the size of the smallest time window within which* n *or more events may occur*

$$\delta^-(n) \quad : \quad \mathbb{N}^+ \mapsto \mathbb{R} \tag{3.1}$$

A maximum event distance function $\delta^+(n)$ *is defined by the size of the largest time window within which less than* n *events may occur*

$$\delta^+(n) \quad : \quad \mathbb{N}^+ \mapsto \mathbb{R} \tag{3.2}$$

where $\mathbb{N}^+$ *is the set of positive natural numbers and* $\mathbb{R}$ *the set of real values.*

Depending on the intention and nature of a performance analysis, the event distance function may not be the most convenient parameter. For example, the busy window approach to derive the response time (as in [Tin94b]) relies on knowing the load imposed by higher priority tasks in order to model the preemption interference (and the distance between activations to model the arrival times after the critical instance). For convenience in such applications, we formally define the *event load function*:

Definition 3.4 (Event Load Functions)**.** *An upper (lower) event load function* $\eta^+(\Delta t)$ *(*$\eta^-(w)$*) is defined by the maximum (minimum) number of events in an event stream that may arrive in a time window of size* Δt.

$$\eta^+(\Delta t) \quad : \quad \mathbb{R}^+ \mapsto \mathbb{N} \tag{3.3}$$

$$\eta^-(\Delta t) \quad : \quad \mathbb{R}^+ \mapsto \mathbb{N} \tag{3.4}$$

when $\mathbb{N}$ *is the set of natural numbers and* $\mathbb{R}^+$ *the set of positive real values.*

The event load functions can straight-forwardly be derived from the event distance functions as follows, and thus has not been explicitly included in the definition of the generalized load event model (Definition 3.2).

$$\eta^+(\Delta t) = \max_{n\geq 1, n\in\mathbb{N}}\{n \mid \delta^-(n) \leq \Delta t\} \tag{3.5}$$

$$\eta^-(\Delta t) = \min_{n\geq 1, n\in\mathbb{N}}\{n \mid \delta^+(n) < \Delta t\} \tag{3.6}$$

Also, when the event load functions are known, the event distance function can be conservatively derived. This can be done with the following conversion:

$$\delta^-(n) = \inf_{\Delta t\geq 0, \Delta t\in\mathbb{R}}\{\Delta t \mid \eta^+(\Delta t) \geq n\} \tag{3.7}$$

$$\delta^+(n) = \sup_{\Delta t\geq 0, \Delta t\in\mathbb{R}}\{\Delta t \mid \eta^-(\Delta t) < n\} \tag{3.8}$$

Figure 3.1 illustrates that both representations contain the same information. Due to the constant segments in each function (i.e. the *eta* function is not bijective), they are however not the mathematical inverse of each other. Le Boudec et al. [LeB01] have coined the term "pseudo-inverse" for this relationship.

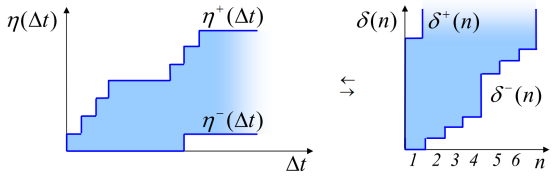

Figure 3.1: Pseudo-inverse Event Model Representations.

3.3 Properties of the Generalized Load Event Model

The system analysis procedure provided in Chapter 2 demands several properties of the analysis parameters in order to ensure the correct convergence. We show in this section, that these properties are fulfilled for the domain of the generalized load event models.

Firstly, we can say that the generalized load event models allow conservatively capturing properties of the modeled event stream according to Definition 2.1. By its definition, it provides bounds on the minimum and maximum distance between the occurrence of any n events. It is conservative if this property holds for the modeled event stream.

3.3.1 General Load Event Models form a Complete Partial Order

Condition 2.5 demands that the domain of each model parameter forms a complete partial order. This the case for the domain of load event models if there is a partial order defined between instances of this event model and there is a smallest as well as a largest element.

Event models can generally be ordered by the set of behaviors that they encompass. For this, we define the comparator "$\sqsupseteq$", read "more generic" that implies that a "more generic event model" encompasses every behavior that is contained in another. In the case of the generalized load event models, this comparator can be defined on the basis of the distance between events, meaning "no less events in any time interval":

Definition 3.5 (Load Event Model Comparator)**.** *A load event model $\eta_1^+ \in \mathbb{E}$ is* more generic *than another load event model $\eta_2^+ \in \mathbb{E}$, $\eta_1^+ \sqsupseteq \eta_2^+$, if and only if for the respective load functions we have*

$$\forall \Delta t > 0 : \quad \eta_1^+(\Delta t) \geq \eta_2^+(\Delta t) \tag{3.9}$$

$$\wedge \quad \eta_1^-(\Delta t) \leq \eta_2^-(\Delta t) \tag{3.10}$$

or, with respect to the event distance functions we have

$$\forall n \geq 1 : \quad \delta_1^-(n) \leq \delta_2^-(n) \tag{3.11}$$

$$\wedge \quad \delta_1^+(n) \geq \delta_2^+(n). \tag{3.12}$$

From Definition 3.5, one can deduce a largest "$\top$" and a least element "$\bot$" of the load event model domain. Let the event model $\top \in \mathbb{E}$ be defined by the event distance functions $\delta_\top^-$ and $\delta_\top^+$, and the event model $\bot \in \mathbb{E}$ be defined by the event distance functions $\delta_\bot^-$ and $\delta_\bot^+$, with the following properties:

$$\forall n \geq 1 : \quad \delta_\top^-(n) = 0, \delta_\top^+(n) = \infty \tag{3.13}$$

$$\delta_\bot^-(n) = \infty, \delta_\bot^+(n) = 0 \tag{3.14}$$

Then we can say, that any event model $e \in \mathbb{E}$ is more generic than $\bot$, and $\top$ is more generic than e:

$$\forall e \in \mathbb{E} : \top \sqsupseteq e \wedge \bot \sqsubseteq e. \tag{3.15}$$

Consequently, $< \mathbb{E}, \sqsupseteq >$ defines a complete partial order, fulfilling the requirement from Condition 2.5. In this thesis we are mainly concerned about the case of high load, which is determined by δ^- (or η^+). The comparison of event models is than reduced to equation (3.11).

3.3.2 Reconstruction of Incomplete Event Stream Information

The generalized load event model concept encompasses any event model that provides the event load or event distance functions. From the perspective of efficient analysis, one is interested in a compact parameterization of the event model. For this purpose [Ric02b] has introduced the *standard event model* concept, which resorts to only three parameters (average distance, jitter, and minimum distance). This model is efficient and already captures a large variety of typical event stream behaviors in real-time systems. On the other end of the spectrum, with high accuracy but limited computing efficiency, one can conceive a table of distances for each number of n events. This of course would have to be infinite in order to comply with Definition 3.2. In between these extremes lies the event model concept of [Wan06a], where event models are represented as an initial "aperiodic" part that can capture a critical transient phase in great detail, and a "periodic" part that than holds for all time intervals starting from a certain size. A similar effect can also be achieved with the event models proposed in [Gre93b].

A complex event model description in combination with an extensive analysis could potentially lead to large analysis times. Therefore, it would be desirable to have a dynamic trade-off analysis speed with accuracy. To achieve this, we now show that it is not necessary to completely specify (and compute) the event distances — and still be able to derive conservative analysis results. This can be achieved by extrapolating information about the timing of events from only partial information. Inspired by the the "super-additive closure" of functions as introduced in [LeB01], we suggest the use of the "super-additive continuation" of the minimum event distance function. Correspondingly, a "sub-additive continuation" can also be defined for the maximum event distance function.

Theorem 3.1 (Subadditive Continuation)**.** *Any minimum event distance function $\delta^-(n)$ known for $n < N^k$ can be conservatively approximated for $n \geq N^k$, $n \leq 2N^k$ as follows:*

$$\delta^-(n) \quad = \quad \min_{0<j<n-1}\{\delta^-(n-j) + \delta^-(j+1)\} \tag{3.16}$$

Note that for practical reasons, only the values $0 < j < \lceil n/2 \rceil$ need to be actually checked, as for larger j, the operands of the δ-functions repeat.

Proof. The proof is by induction. As stated in the assumptions, all given values $\delta^-(n)$ for $n < N^k$ are correct conservative estimates of the minimal distances between n events. We will now show the induction step: if (3.16) is true for all n with $n < N < N^k$, it is also true for $n = N$.

This sub-proof is by contradiction. Assume there is a number of events $n = N$ for which the actual distance $d(n)$ is smaller than stated in Equation 3.16.

$$\exists n : d(n) < \min_{0<j<n-1}\{\delta^-(n-j) + \delta^-(j+1)\} \tag{3.17}$$

This implies that there is a j with $0 < j < n-1$ for which the right hand side of Equation 3.17 is minimized.

$$\exists n, j \mid 0 < j < n-1 : d(n) < \delta^-(n-j) + \delta^-(j+1) \tag{3.18}$$

Both $n-j$ and $j+1$ are smaller than N^k, and thus it is known that $\delta^-(n-j)$ and $\delta^-(j+1)$ are conservative estimates of the minimum distance between $n-j$, $j+1$ events respectively. These $n-j+j+1-1$ events can not occur closer to each other than in the scenario depicted in Figure 3.2 where the last of the "left" $n-j$ events is also the first of the "right" $j+1$ events. Even in this scenario n events will still be separated by the sum of $\delta^-(n-j)$ and $\delta^-(j+1)$, which contradicts equation(3.18).

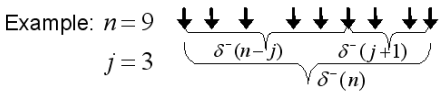

Figure 3.2: Composition of minimum distances.

This contradiction proves the induction step and leads us to accept the lemma as true. □

This continuation procedure event allows performing an analysis event when the description of the input event models are incomplete. When at least $\delta^-(2)$ is known, the distance between any larger number of events can be extrapolated; with increasing accuracy when the distance between a larger amount of events has been provided. This provides us with to the ability to abort any analysis of a task output event model at any time after the minimum distance between two events is known.

One then also does not need to explicitly reason about the periodically repeating part of an event model (e.g. of [Ric02b] or [Wan06a]) , but the procedure can be used to set the parameters of the event model correspondingly.

In many cases, the sub-additive continuation can be exact where a simple periodic continuation would be inaccurate. For example, an event stream with periodic bursts is precisely reconstructed when the distance of the events in one hyperperiod is known. In a simple periodic continuation, i.e. through the linear bound on the event pattern, this behavior would only be conservatively over-approximated.

3.4 Resource Models

We will now turn to the problem of deriving the timing properties of events produced inside the system from the timing of events by which the system is externally stimulated. The procedure presented in Chapter 2 suggests a decomposition of the complete system into subsystems ("components"), between which the interaction is

iteratively refined. A prerequisite for this procedure is thus the ability to derive the timing of events produced by a component based on the analysis parameters at its inputs.

Similar to the event model concept, which allows to abstract irrelevant details of the event behavior, one does at a certain stage of the analysis not want to be concerned about the details of the components state and behavior, in particular if it has little or no impact on the resulting timing. Various research efforts have provided abstractions for the behavior of a subsystem, with different degrees of accuracy and complexity. This section reviews the most important metrics, and highlights the need for an new efficient model.

3.4.1 Task Response Time

The simplest conceivable conservative scheduling-aware timing abstraction of a processing component is its worst-case response time, which represents the maximum amount of time between the arrival of an event (i.e. activation of a task) and the completion of the corresponding task instance, taking into account any possible execution delays and scheduling interference.

This bound is the underlying resource abstraction in the compositional analysis approaches in [Gre93a] and [Ric02b]. It is also the timing annotation of choice for timed actor oriented execution models that allow abstracting the properties of the underlying scheduling and execution platform [Bek04]. In other actor-oriented analyses, the actors may be agnostic to the underlying scheduling and hold only their respective *execution time* [Lee03]; in this case it is difficult to talk about a sub-system abstraction.

In [Ric02b, Ric04], which is focused on the standard event models with the three parameters period, jitter, and minimum distance, the output event model of a task is computed by increasing the jitter parameter of the input event model by the difference between the task's best- and worst-case response times. As the worst-case response time can be determined with classical (single-)processor scheduling theory, this method allows addressing a large set of scheduling and arbitration policies. However, relying on the response time bounds alone yields overly conservative results in cases where the worst-case response time is only an effect of a transient overload situation. The metric will then enable only insufficient derivations of tasks output event models, and end-to-end path latencies.

The restriction to standard event models can be easily dropped. The generalized load event model at the output of a task with an input event model given by $\delta_{in}^{-}(n)$ and $\delta_{in}^{+}(n)$ can be derived as follows:

$$\delta_{out}^{-}(n) = \max\{\delta_{in}^{-}(n) - J^{resp}, \delta_{out}^{-}(n-1) + d^{min}\} \tag{3.19}$$

$$\delta_{out}^{+}(n) = \delta_{in}^{+}(n) + J^{resp} \tag{3.20}$$

where $J^{resp} = R^{max} - R^{min}$ is the task's response time jitter, i.e. the difference between its best and worst-case response time (see also [Sch09c, Rox08]).

3.4.2 Continuous Service Bounds

The problem with the response-time based approach is that it represents the local behavior with respect to a specific combination of a) the resource service b) the task behavior, and c) the actual event workload. This condensation to a single parameter has difficulty to accurately capture the behavior over larger intervals (this will be detailed in Section 3.4.4). Dedicated modeling of the resource service independently of the task workload promises to deliver more insight into such setups.

In Network Calculus [LeB01] and the Real-Time Calculus which is based on it [Thi00, Cha03a], the local resource behavior is modeled with so called *service curves*. A service curve represents the execution time provided to the processing of tasks activated by events in a stream. Such a resource service curve is depicted in Figure 3.3a, for minimum ($\beta^-(\Delta t)$) and maximum supplied service ($\beta^+(\Delta t)$) within a time window of given size Δt. Appropriate service curves have been provided e.g. for static priority preemptive scheduling, EDF, TDMA, and others [Cha03a], and also hierarchical scheduling [Wan06c]. However, specific resource service curves are difficult to derive in cases where the actual amount of events has an impact on the achievable service. This is the case for example for preemption-related overhead such as context switch time or cache-related preemption delay, or blocking effects including non-preemptive and collaborative scheduling.

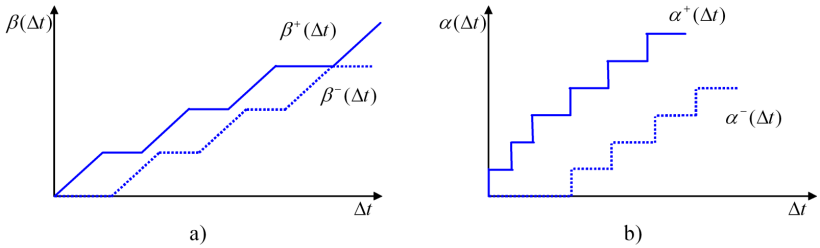

Figure 3.3: Resource Service and Event Arrival Curves in Real-Time Calculus [Cha03a].

In network calculus, each stream of incoming events is described by a general *arrival curve* (see Section 3.2), which is a continuous sub-additive function $\alpha(\Delta t)$ that describes the maximum (and also minimum) number of events that may arrive in *any* time window of size Δt (see Figure 3.3b). The arrival curves can be translated into *demand bound functions* [Wan06a] that provide the resulting amount of computation implied by the task that is activated by the respective events. Differences in compu-

tational load implied by different tasks can be modeled by scaling the arrival curve by a constant factor representing the task execution time. Just like the general event model concept, the arrival curve interface naturally allows to capture the behavior of more specific event models, such as the "period with jitter" event model proposed in [Ric02b] or the pattern description of [Gre93b]. A number of operations have been defined in network calculus to allow the derivation of traffic descriptions within the system as well as buffer and latency information. For example, the *outgoing arrival curve* a^+_{out} can be derived from the incoming arrival curve a^+_{in} and the minimum and maximum amount of provided resource service β^+ and β^- according to the following equation (cited from [Cha03a]):

$$\begin{aligned} \alpha^+_{out}(\Delta t) \quad = \quad & \min\{\inf_{0\leq\mu\leq\Delta t}\{\sup_{\lambda\geq 0}\{\alpha^+_{in}(\mu+\lambda)-\beta^+(\lambda)\} \\ & +\beta^-(\Delta-\mu)\}, \beta^-(\Delta)\} \end{aligned} \tag{3.21}$$

As this approach allows to express a larger variety of service and event arrival patterns, the resulting event estimates are more precise when compared to the accumulation of response time jitter [Per08].

This procedure can however be computationally intensive as it derives output event models and remaining resource capacity by folding operations in the continuous time domain. For this reason practical simplifications have been suggested (e.g. stepwise evaluation [Cha05b] or finite models of event streams [LT02, Wan06a]). Instead of relying on arbitrary continuous service and arrival curves, [Sti98] introduces the concept of latency-rate ($\mathcal{LR}$) servers to represent the resource service with only two parameters (based on [Cru91a, Cru91b]). The key benefit is, that sequences of $\mathcal{LR}$-servers can be represented as a single node which provides a very simple analyzability. Many actual schedulers can be abstracted by the model, however the behavior of schedulers that provide changing service rates has to conservatively approximated (this is the case for example for the service provided in a round-robin scheduler in which competing time slots are only occupied for a transient phase).

The latency-rate abstraction has also be used to consider such schedulers in dataflow models. In [Wig07], the run-time scheduling delay is captured by a set of actors in a dataflow graph that exhibit the same worst-case behavior as specified for latency-rate servers. In [Wig09] the concept has been applied to budget schedulers, which provide a minimum share of the resource service per time interval. [Wig09] generalizes the work of [Sti98] in several ways. It enables the analysis of task graphs with an arbitrary topology and allows for buffers with a fixed capacity to influence the temporal behavior. A requirement is that the task graphs have a functional deterministic behavior. Therefore also applications with a multi-rate or cyclo-static behavior can be analyzed. This work has been further generalized in [Sta09a] where a dataflow model is presented to capture more accurately than possible with the latency-rate server model the behavior of schedulers with a changing minimum service provision.

3.4.3 State-Based Resource Models

More sophisticated modeling of a resource's timing behavior is possible with state-based models which take into account that the resource may be in a number of different states at the time of the task activation, and that depending on this state, different response times may follow. The state may be defined by a particular processing mode (such as a high-speed mode in overload situations, and a regular mode in the average case) or externally defined (e.g. dependent on the current power situation).

It is not uncommon that the resource's state depends on the history of preceding events. For example, the buffer fill level at the time of the arrival of an event is the result of the distance to preceding tokens. A resource may decide to reject an incoming job depending on this buffer state. Such behavior has been modeled in [Bou09]. A general method to capture the possible sub-system's states is by expressing it as a "timed automaton" [Alu90, Nor99, Hen06a]. Timed automata allow expressing the possible states and transitions between them together with a set of clocks to keep track of the timing relations. However, the system complexity grows exponentially with the number of clocks, which makes analyses of more complex systems difficult to impossible. A similar approach is chosen in [Cha05a], where the sate progressions of a resource and the arriving workload can be captured with so called "event-count automata".

Another possibility to capture the system states in more detail are dataflow graphs, such as timed petri nets (TPN) or synchronous dataflow (SDF) graphs [Lee87] in which the current state is modeled through the placement of tokens in the graph. However, this implies that a subsystem may be in an exponential number of possible states at the time of the arrival of external event. Such graphs exhibit monotonic properties in the time-domain ("the earlier arrival of one event may not lead to the later occurrence of another" [Bek04]). However, when the behavior in transient corner-cases is of interest (in particular to determine the maximum transient load), it is difficult to determine the most critical initial state and state transitions (see also [Sch07]).

Due to the constrained scalability of the state-based analysis approaches, efforts have been invested to combine them with state-less models. In [Lam09] the real-time calculus has been combined with timed-automata models, and in [Pha07] with the analysis based on event-count automata. In [Sch07] compositional performance analysis based on event streams was coupled with an analysis of synchronous dataflow graphs.

To a certain extend, the state-dependent behavior can also be "projected" into state-less models. For example, the resource may iterate over a set of cyclo-static execution times. Although the resource will react to different events with different execution times, there is a worst-case subsequence that can be derived a priori (such as in [Mok97, Jer04]). In this case, the actual state is then not relevant anymore as the resulting worst-case timing can be bounded. This approach is highly beneficial for

the resulting computation time, as it reduces the analyzed state space to the relevant candidates.

3.4.4 The Need for a new Model

The present resource models and event propagation functions represent different trade-offs between accuracy and complexity. While the response-time abstraction is very simple, and also accurate for many typical scenarios, it is inadequate to reason about the correlation between the arrival time of an event with respect to its predecessors and the resulting response time. This correlation is much better considered by network calculus, but the folding of functions in the continuous time domain implies long computation times. With specific service models such as the latency-rate servers, the analyzability improves at the cost of accuracy and scope. The analysis complexity further grows when state-based models are involved. In the next section, a new abstraction of resource timing is introduced that combines the advantages. This model, the *multiple event busy time model*, is a direct extension of the classical busy window approaches such as [Jos86, Har87, Leh90, Tin94b] that symbolically derive worst-case task sequences considering context switches, blocking, release offsets, or mutual dependencies. We generalize this concept by explicitly tracking the individual events in the busy window and allow idle times between busy periods when investigating the event streams' timing behavior. Only the discrete points represented by the busy time function are relevant to the analysis, so that this procedure can be expected to be more efficient than many continuous time methods. We then use the new resource model in Section 3.6 to derive event models at the task outputs, that consider transient load situations and also allow for idle times between them. The model also later serves as the basis for a general procedure to derive end-to-end latencies that involve multiple resources in Chapter 4.

3.5 Multiple Event Busy Time Model

This section elaborates the concept of the busy period that is used in most response-time analyses that use the windowing technique to determine a task's worst-case response time. For this, Lehoczky defines the busy period in [Leh90] (based on [Jos86]) as the time interval during which tasks above a certain priority level are being processed, such that the resource is idle immediately before and after. He goes on to show that in static priority preemptive scheduling the worst-case response time is experienced by a task activation within a busy period that begins with a "critical instant" (as originally defined by [Liu73]).

The *multiple event busy time* function is a generalization of this concept. The busy time function represents the amount of time necessary to process a certain number of events that arrive within the same busy window. For example, $B^+(1)$ is the maximum busy window inflicted by a single event that arrives after the previous was finished. $B^+(2)$ is the maximum busy window size that is spanned by two events, where the

second arrives before the first is finished. We also call this scenario "2 task instances are coinciding", meaning that "*only* 2 task instances are coinciding".

Definition 3.6 (Multiple Event Busy Time)**.** *The maximum (minimum) q-event busy time $B_i^+(q)$ ($B_i^-(q)$) of a task τ_i is given by the maximum (minimum) time it may take instances of τ_i to process q events, if all but the first of the q events arrive before the preceding is finished.*

The busy time function *delivers $B_i^+(q)$ for any $q \in \mathbb{N}^+$.*

For illustration consider the example schedule in Figure 3.4. The first activation of task T3 experiences a critical instant scenario for static priority preemptive scheduling: all higher priority tasks are activated at the same time and as early as possible thereafter. This leads to a worst case busy time of $B_{T3}^+(1) = 15$, which is the sum of the involved core execution times. The next activation (arriving at time 11) is processed subsequently, and finished no later than $B_{T3}^+(2) = 22$.

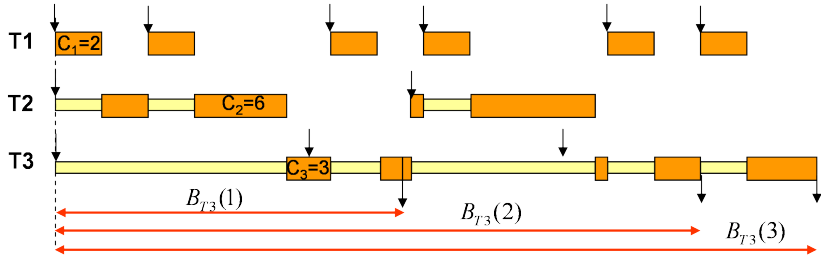

Figure 3.4: Multiple Event Busy Times.

We follow the assumption of the previous work introduced in Section 3.4 and assume that every task processes its events in order (prioritization can be modeled by separate tasks).

By the above definition, the busy time contains all effects that can delay the finishing of the task activations. This deviates from the definition in [Leh90], where the resource is explicitly occupied (not idle). The multiple event busy time includes not only the task's execution times and the interference by other tasks mapped to the same resource, but also inter-task communication, context switch overhead, and any other delaying factors (if present). In particular, we show in Chapter 6, how the use of a shared resource will impact a task's the busy time.

This busy time concept has been implicitly used in many previous scheduling analyses that rely on the windowing technique [Jos86, Leh90, Tin94b, Li98, Rac07]. During the calculation of the worst-case response time, finishing times of different task instances are calculated — in a worst-case scenario this corresponds to busy times as defined

above. Thus, in this case the calculation of the busy times comes at no additional computational costs. But despite its obvious similarity to the previous use of the busy period, the above definition is notably different in the following aspects:

- The multiple event busy time does not depend on the actual activation pattern of the investigated task (T3 in the example). This fully decouples the resource model from the event model.
- The multiple event busy time does not imply that the resource is not idle before the events have been processed (but this will be the case for work conserving schedulers as long as no external resources are used).

The busy-time function can also handle tasks with variable execution times as will be demonstrated in Section 3.5.2. We will actually see in Section 3.5.4 that it can be derived for any scheduling policy, for which either a service curve exists, or a the worst-case response time can be computed.

3.5.1 Deriving a Task's Worst-Case Response Time from its Multiple Event Busy Time

When a task's busy time function is known, and the task is subjected to a stream of activating events with a minimum distance of $\delta_i^-(n)$, then its worst-case response time can be straight-forwardly computed. The following lemma corresponds to Tindell's implicit reasoning in [Tin94b].

Lemma 3.2. *A task τ_i with multiple event busy time function $B_i^+(q)$ that is activated by events with a minimum distance of $\delta_i^-(q)$ has a response time smaller than R_i:*

$$R_i = \max_{0<q}(B_i^+(q) - \delta_i^-(q)) \tag{3.22}$$

We omit the proof of this lemma, because it represents a special case of the worst-case path latency with a path length of 1, which is formally investigated in Chapter 4.

3.5.2 Application to Static Priority Preemptive Scheduling

To demonstrate the feasibility of the concept, Lemma 3.3 provides the multiple event busy time for static priority preemptive scheduling of independent tasks (on the basis of [Tin94b]).

Lemma 3.3. *The multiple event busy time $B_i^+(q)$ for a task τ_i under static priority preemptive scheduling with independent tasks that do not perform accesses on external shared resources is given by*

$$B_i^+(q) = q \cdot C_i + \sum_{j \in hp(i)} (\eta_j^+(B_i^+(q)) \cdot C_j \tag{3.23}$$

where

- C_i, C_j *is the maximum core execution time of task* τ_i, τ_j.
- $hp(i)$ *is the set of tasks with higher priority than* τ_i.
- $\eta_i^+(\Delta t)$ *is the maximum number of events that lead to an activation of task* τ_i *in a time window of size* Δt.

Proof. The proof follows Tindell's argumentation in [Tin94b]. The q coinciding activations of task τ_i are finished when their combined workload has been processed (first term $q \cdot C_i$). The processing of this workload in the given scheduler can be delayed only by higher priority task activations. During the processing of the q events, $B_i^+(q)$, the maximum number of higher priority task activations is given by $\sum_{j \in hp(i)} (\eta_j^+(B_i^+(q))$ and their combined workload can not be larger than the second term. Shifting the task activations backward or forward in time can not lead to a larger combined work load during $B_i^+(q)$. □

As $B_i^+(q)$ is used on both sides of Equation 3.23 no direct solution is available. However, the right hand side is monotonic with respect to $B_i^+(q)$, and thus the fixed-point can be found through iteration (very similar to [Jos86]). Obviously the approach supports arbitrary deadlines.

Similarly, other more sophisticated windowing based analyses can be straight-forwardly extended to produce the busy time function. This allows to consider e.g. blocking due to shared resources [Raj91], task preemption costs [Bur95], task offsets [Pal98], or variable task execution times [Mok97]. For example, when variable task execution times ("context-dependent execution times") are known, the term $q \cdot C_i$ in Equation (3.23) can be replaced by the worst-case workload implied by a set of consecutive task instances such as provided in [Jer04, Max04]. If offsets are taken into account (such as in the worst-case response time analysis of [Hen06b]), it may be that the busy time of n events does not occur in the same scenario in which the response time is maximized. In this case, the busy time for each n is given by the maximum over all scenarios that are checked during original the analysis.

There may be other scheduling overheads that can influence the task's response time, e.g. caused by context switches, pipeline flushes, non-preemptable code sections, or general operating system operations. These overheads are not the concern of this chapter, but they can mostly be covered by adding additional terms to the workload equation (as we will se in Chapter 6.

3.5.3 Application to Time-Driven Scheduling

The multiple event busy time can also be provided for time-driven scheduling policies, for example TDMA. Under TDMA, the processor is cyclically assigned to the tasks, each for a fixed duration (*time slot*). If a task has nothing to execute, the time slot passes unused. If a task overuses its budget, it will be preempted, and the response

time of the other tasks will not be affected. This highly deterministic behavior makes TDMA attractive for the use in safety-critical applications.

Theorem 3.4 (Multiple Event Busy Time for TDMA Scheduling)**.** *Under time-driven multiple access scheduling, the multiple event busy time of q instances of a task τ_i is given by*

$$B_i^+(q) = q \cdot C_i + \sum_{j \in T \setminus \{i\}} \left\lceil \frac{q \cdot C_i}{t_i} \right\rceil \cdot t_j \tag{3.24}$$

where

- *C_i is the maximum core execution time of task τ_i.*
- *T is the set of scheduled tasks*
- *t_i, t_j are the slot sizes assigned to task τ_i, τ_j*

Proof. Firstly, q instances of task τ_i are completed only when they have received a total time of $q \cdot C_i$ on the processor. For this, the task has to occupy $\lceil (q \cdot C_i)/t_i \rceil$ of its time slots at least in part. Before the assignment of the first time slot, every other task may receive at most one time slot, which may not take longer than $\sum_{j \in T \setminus \{i\}} t_j$. The same is true between the successive $\lceil q \cdot C_i/t_i \rceil - 1$ time slots of τ_i. Thus in total, the waiting time for τ_i's time slots is as stated in the theorem. □

3.5.4 Alternative Methods for Deriving the Busy Time Function

In some setups, analysis time may be crucial. For example, during the design space exploration phase in early design phases rough performance estimates can be sufficient to guide the exploration (as in [Ham06]). One may find it unnecessarily time-consuming to compute the multiple event busy time function to characterize the task timing for large numbers of overlapping task instances. Also in another scenario, when admission control is to be performed at run-time (as proposed in [Ste06]) the available hardware usually supplies only limited computing power.

For these use-cases the analysis effort can be reduced with the help of approximations as provided in this section. The speed-up is bought by a loss in accuracy. A reasonable exploration procedure would thus be to begin an exploration at a fast analysis speed, and only iterate on the most promising design choices with a higher detail.

3.5.4.1 Sub-Additive Continuation of Busy Time Function

Quite like the "super-additive continuation" of incomplete event models proposed in Section 3.3.2, a continuation function can also be defined for the multiple event busy time that allows to extrapolate the function beyond values that have been explicitly calculated.

The key idea is that if it is known that p and q coinciding instances of a task will be completed no later than $B^+(p)$ and $B^+(q)$, one can deduce that $p+q$ coinciding instances must be completed no later than $B^+(p+q)$.

This reasoning allows to extrapolate the busy time function for larger q from its value for small q. A loss of precision can be expected, because the combination of the two "busy time segments" possibly implies two "critical instants", while in fact only one critical instant can be observed in the complete interval.

Theorem 3.5. *Any multiple event busy time function $B^+(q)$ known for $q \leq N$ can be conservatively approximated for $q > N$ as follows:*

$$B^+(q) \leq \min_{0 \leq p < q} \{B^+(q-p) + B^+(p)\}$$

Proof. The proof is by contradiction. Assume there q coinciding activations will span a busy time $B(q)$ that is larger than given in the theorem. This implies there must be an p for which the above equation is minimized and for which we have:

$$\exists p, 0 \leq p < n : B(q) > B^+(q-p) + B^+(p)$$

This however can not be, because according to the definition of the multiple event busy time the first $q-p$ events must be processed within a time no larger than $B^+(q-p)$, and the last p events must be processed in a time no larger than $B^+(p)$ and thus $B(q) \leq B^+(q-p) + B^+(p)$. □

3.5.4.2 Approximating the Busy Time Function with the Task's Response Time

Theorem 3.5 also indicates a fall-back procedure for the case where no knowledge of the busy time exists for a specific scheduler, but one knows how to derive the task's worst-case response time with an analysis. Such response time analyses exist for virtually every real-time scheduling policy.

Let the task's worst-case response time be bounded by R_i. This implies that *any* task instance is finished no later than R_i after the corresponding event has arrived. Consequently, we have

$$B_i^+(1) \leq R_i \tag{3.25}$$

With this, and Theorem 3.5, one can quickly conclude that

$$B_i^+(q) \leq R_i \cdot q \tag{3.26}$$

Consequently, the busy time function can be approximated with the best known worst-case response time estimate for any scheduling policy for which the response time can be provided.

3.5.4.3 Derivation of the Multiple Event Busy Time from Service Curves

The multiple event busy time function can also be straight-forwardly derived from the service curves from Network Calculus [LeB01] or Real-Time Calculus [Cha03a]. As these functions express the amount of computation time that is guaranteed to be provided for a specific task, one can deduce from the task's worst-case execution time (WCET) the maximum time before successive activations are finished.

Theorem 3.6. *Let a task τ_i with worst-case execution time C_i be executed on a resource on which it receives a service of at least $\beta^-(\Delta t)$ in any time window of size Δt, than its multiple event busy time function is as follows*

$$B_i^+(q) = \min_{0<\Delta t} \{\beta^-(\Delta t) \geq q \cdot C_i\} \tag{3.27}$$

Proof. q coinciding instances of τ_i will be completed when they were assigned the resource for a total time of $q \cdot C_i$. This is the case for any time window Δt for which $\beta^-(\Delta t) \geq q \cdot C_i$. The processing of the q events may then not take longer than the smallest time window size for which this is fulfilled. □

Figure 3.5 shows the relation between the service curve and the corresponding multiple-event busy time function.

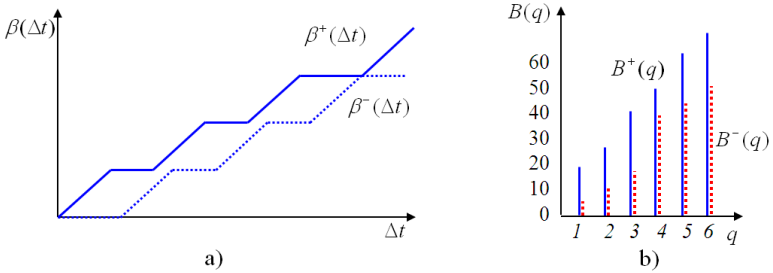

Figure 3.5: a) Service curve and b) corresponding Multiple Event Busy Time Function.

The service curves of [Cha03a] thus provide more information about the resource behavior than the multiple event busy time function, but at a higher cost. The drawback of relying on the service curve is twofold: Firstly, classical scheduling theory can not simply be reused. This makes it difficult to include the diversity of overheads and extensions as are available for the busy window technique (see Section 6.1.1 for a discussion on this topic). Secondly, and more important in theory, the accompanying analysis procedures are computationally intensive folding operations in the continuous

time domain (see Section 3.4.2). Most of the service curve information is actually irrelevant for deriving event flow. For the performance analysis it is usually not relevant to know when a task activation is "half-finished" as opposed to when it is fully finished.

3.6 Deriving Output Event Models

A core element of the compositional analysis presented in Section 2 is the procedure of deriving a task's output event model from the task's input event model and local resource model. This procedure is called the "propagation" of event models in [Ric04].

Based on the multiple-event busy time introduced in Section 3.5, a new propagation procedure is provided in this section that improves previous approaches in several directions:

- The class of supported event models is extended from "Standard Event Models" to the general event model interface (Definition 3.2).
- The achieved accuracy is increased by relying on the multiple event busy time function to represent the task's timing instead of its worst-case response time.
- The computational effort remains bounded by investigating a discrete set of relevant candidates.

The generality of this method with respect to supported event and resource models allows a large applicability and accuracy in different domains (which is demonstrated in Section 3.7 for a small system and again in Chapter 8 for larger setups).

3.6.1 Derivation of Minimum Event Distances

As described above, a task's output event model was derived in previous work on the basis of its worst-case response time. However an event's arrival time and its resulting response time are actually correlated due to the current processing backlog.

For an example, consider finding the minimum distance between the production of two events. Let all events be numbered according to the sequence of their occurrence — events occurring later receive higher numbers. Then let the later of the two events be event 0 and the preceding event be event -1. In any case, event 0 will be processed no sooner than $B^-(1)$ after its arrival. So the problem can be reduced to finding the maximum finishing time of event -1.

The production time of event -1 depends on the state of the resource at the time of its arrival at the task input. If the resource has completed all preceding events at that time, then -1 will be finished no later than $B^+(1)$ after it has arrived (see Scenario a) in Figure 3.6). If a previous event, number -2, is still being processed, the processing of event -1 contributes to the ongoing busy interval. This interval is then finished no later than $B^+(2)$ after the arrival of the preceding event -2 (Scenario 3.6b). Thus, the maximum finishing time is actually the maximum over the end times of the busy intervals started by all previous events.

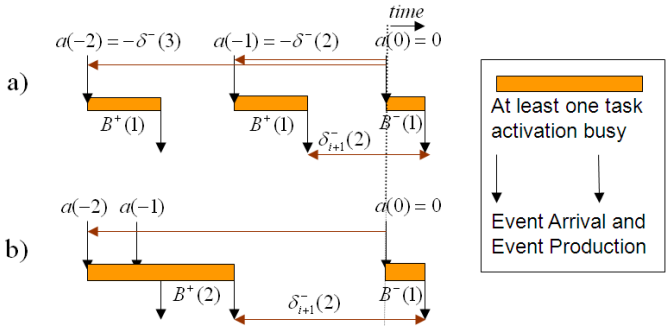

Figure 3.6: a) Busy times of preceding events may not overlap b) may overlap.

The following lemmata allow us to formally reason about the distance between the production of events at the output of an arbitrary task τ_i. For this, we first establish the maximum exit time $e_i(n)$ of any event n with respect to its arrival time $a_i(n)$, and the arrival times of its predecessors (Lemma 3.7). We then determine the constraints that are imposed on the distance between arriving events owed to their comprising event model (Lemma 3.8). This allows us to bound the number of scenarios that have to be investigated in order to determine the maximum exit time (in Lemma 3.9). Finally, based on the reasoning in the above example, we provide Theorem 3.11 that allows computing the minimum distance between the events produced by task τ_i.

Firstly, the following Lemma provides the maximum exit time of an arbitrary event that is produced by τ_i. The event that leads to the activation of a task instance, and the event that is produced by the same instance, are denoted with the same number (this notion will be further formalized in Chapter 4, where the latency of events along a path is examined).

Lemma 3.7. *The exit time $e_i(n)$ of any event n produced by task τ_i is bounded by*

$$e_i(n) \leq \max_{k \geq 0} \{a_i(n-k) + B_i^+(k+1)\} \tag{3.28}$$

where

- *$a_i(n-k)$ is the arrival time of the k-th event before event n.*
- *$B_i^+(k+1)$ is the maximum multiple event busy time for $k+1$ events to be processed by τ_i.*

Proof. The output event n is produced at time $e_i(n)$ by the task instance of τ_i that is activated by the event n that arrives at time $a_i(n)$. There are two cases: Either the previous activation of τ_i is already finished when the input event arrives, or it

is not. We assume that an event arriving at the very instant at which the previous activation terminates does not fall into the same busy interval.

Case 1: $a_i(n) \geq e_i(n-1)$ (previous activation is finished). In this case, following Definition 3.6, the activation n is finished no later than $a_i(n) + B_i(1)$. Case 2: $a_i(n) < e_i(n-1)$ (previous activation is not finished). The input event n arrives while at least one previous event has not been produced. Let k be the number of events that have not been processed. Thus, $e_i(n-k-1) \leq a_i(n)$ and

$$a_i(n) < e_i(n-k). \tag{3.29}$$

In this case, the multiple event busy time bounds the time at which the busy interval that was started by event $n-k$ (and to which the event n now contributes to) is over:

$$\Rightarrow \qquad e_i(n) \leq a_i(n-k) + B_i(k+1) \tag{3.30}$$

Thus, Case 1 can be seen as a special case of Case 2, for $k = 0$. In general, it is not possible to determine in advance which conditions are fulfilled (i.e. how many events are still unprocessed at the time of $-n+1$'s arrival), because this depends on the run-time unfolding of events. But it remains conservative to assume the maximum exit time over all possible values of k. The lemma follows. □

The set of backlogged events, $\{k \geq 0\}$ in equation(3.28), can be further constrained by considering that there is a minimum distance between arriving events and the maximal time for their processing. This is provided by the following two lemmata.

Lemma 3.8. *With respect to the arrival of an event p at task τ_i, the event q, arrives at τ_i no later than*

$$a_i(p) \quad \leq \quad \begin{cases} a_i(q) - \delta^-_{i,in}(q-p+1), & \text{if } p < q \\ a_i(q) + \delta^+_{i,in}(p-q+1), & \text{if } p > q \end{cases}$$

where $\delta^-_{i,in}(n)$ is the minimum and $\delta^+_{i,in}(n)$ is the maximum distance between any n events arriving at task τ_i.

Proof. All events leading to activations of task τ_i are constrained by their corresponding event stream, which defines a minimum $\delta^-_{i,in}(n)$ and maximum $\delta^+_{i,in}(n)$ distance between the occurrence of any n events. Thus, if the arrival time of one event p is known, all preceding and successive events have a predefined minimum and maximum distance from this event according to (3.31) and (3.32).

$$\forall p, q \in \mathbb{N}, p < q: \quad a_i(q) - a_i(p) \quad \geq \quad \delta^-_{i,in}(q-p+1) \tag{3.31}$$

$$a_i(q) - a_i(p) \quad \leq \quad \delta^+_{i,in}(q-p+1) \tag{3.32}$$

Thus, for $p < q$, we have

$$a_i(p) \quad \leq \quad a_i(q) - \delta^-_{i,in}(q-p+1) \tag{3.33}$$

and for $p < q$,

$$a_i(p) \quad \leq \quad a_i(q) + \delta^+_{i,in}(p - q + 1) \tag{3.34}$$

□

Lemma 3.9. *When a new event arrives at the input of a task τ_i, the set of the number of events that may be unfinished at this time is bounded by*

$$K \quad = \quad \{k \in \mathbb{N}^+ \mid \delta^-_{i,in}(k+1) < B^+_i(k)\} \tag{3.35}$$

where $\delta^-_{i,in}$ is the minimum distance between events that activate τ_i and B^+_i is the multiple event busy time.

Proof. Condition 3.29 in the proof of Lemma 3.7 is fulfilled when the event n falls into the busy period that was started by event $n-k$, and thus we have $e_i(n-k-1) \leq a_i(n)$ and $a_i(n) < e_i(n-k)$. This can only be the case when

$$a_i(n) < a_i(n-k) + B^+_i(k) \tag{3.36}$$

We know from Lemma 3.8 that $a_i(n) \leq a_i(n-k) + \delta^-_{i,in}(k+1)$, an thus Equation 3.36 can only be fulfilled as long as

$$a_i(n-k) + \delta^-_{i,in}(k+1) \quad < \quad a_i(n-k) + B^+_i(k) \tag{3.37}$$

$$\delta^-_{i,in}(k+1) \quad < \quad B^+_i(k). \tag{3.38}$$

i.e. the minimum distance between the arrival of event n and event $n-k$ is smaller than the maximum busy time that is started by event $n-k$. This reduces condition $\{k \geq 0\}$ to the set stated in the Lemma. □

In a schedulable system, the set defined in Equation 3.35 is finite, because the minimum distance $\delta^-_{i,in}$ between incoming events must eventually be larger than the maximum busy time B^+_i to process these events. The set of candidates can be found numerically, for example by testing this condition for increasing k: In this case, the maximum busy time function will grow sub-additively, while the minimum distance between events at the input grows only super-additively. This implies that the values eventually intersect. In an efficient implementation, this condition can be checked en passant the computation of the task's response time, which is also computed incrementally for an increasing number of coinciding task instances.

From the preceding lemmata, we can construct the latest possible production time of any event.

Corollary 3.10 (Maximum Exit Time)**.** *The exit time $e_i(n)$ of any event n produced by task τ_i is bounded by*

$$e_i(n) \leq \max_{k \in K}\{a_i(n-k) + B_i^+(k+1)\} \tag{3.39}$$

$$K = \{k \in \mathbb{N}^+ \mid \delta_{i,in}^-(k+1) < B_i^+(k)\} \tag{3.40}$$

where a_i and B_i^+ are defined as in Lemma 3.7, and $\delta_{i,in}^-$ is the minimum distance between incoming events.

These considerations allow us to formulate our main theorem, that allows the derivation of the minimum distances at the output of a task with a given input event model and multiple event busy time function. The proof establishes this distance by tracking the difference between the latest production of one event (based on the previous reasoning) and the earliest production of another (based on the minimum multiple event busy time of one event $B_i^-(1)$).

Theorem 3.11. *Given a task τ_i with maximum busy time function $B_i^+(n)$ and minimum busy time $B_i^-(1)$. When the minimum distance between n arriving events is bounded by $\delta_{i,in}^-(n)$ then the minimum distance between n events at the output of this task is given by $\delta_{i,out}^-(n)$ as follows:*

$$\delta_{i,out}^-(n) = \max[0, \min_{k \in K}\{\delta_{i,in}^-(n+k) - B_i^+(k+1)\} + B_i^-(1)] \tag{3.41}$$

$$K = \{k \in \mathbb{N}^+ \mid \delta_{i,in}^-(k+1) \leq B_i^+(k)\} \tag{3.42}$$

Proof. Let the arrival time of an event n at the resource to which task τ_i is mapped be $a_i(n)$ and the time at which the resulting task instance produces an event be $e_i(n)$. The distance between any n events at the output can never be smaller than the minimum time between the production of an event m and the production time of an event q that has been produced $n-1$ events earlier (thus $q = m-(n-1)$). Let $e_i^-(m)$ be the earliest possible production time of event m and $e_i^+(q)$ the latest production time of q. Note that event m is always produced after event q, due to in-order processing. Thus, we have the minimum distance between the production of the corresponding events bounded as follows:

$$\delta_{i,out}^-(n) = \max[0, e_i^-(m) - e_i^+(m-n+1)] \tag{3.43}$$

For the last of these considered events, event m, we can safely assume that it could not be produced earlier than $a_i(m) + B_i^-(1)$.

$$e_i(m) \geq a_i(m) + B_i^-(1) \tag{3.44}$$

Thus, the problem that now remains is to find the maximum production time of event $m-n+1$. The production time of this event, $e_i(m-n+1)$, is given by the time at which the corresponding input event has arrived ($a_i(m-n+1)$) plus the amount of time (B) that its resource was busy processing it. This is bounded according to Corollary 3.10:

$$e_i(m-n+1) \leq \max_{k \in K}\{a_i(m-n+1-k) + B_i^+(k+1)\} \tag{3.45}$$

From Lemma 3.8, we furthermore know that

$$a_i(m-n+1-k) \leq a_i(m) - \delta_{i,in}^-(n+k) \tag{3.46}$$

and thus

$$\begin{aligned} e_i(m-n+1) &\leq \max_{k \in K}\{a_i(m) - \delta_{i,in}^-(n+k) + B_i^+(k+1)\} && (3.47)\\ &= a_i(m) + \max_{k \in K}\{-\delta_{i,in}^-(n+k) + B_i^+(k+1)\} && (3.48)\\ &= a_i(m) - \min_{k \in K}\{\delta_{i,in}^-(n+k) - B_i^+(k+1)\} && (3.49) \end{aligned}$$

and thus Equation 3.43 can be evaluated with Equations 3.44 and 3.47 as follows:

$$\delta_{i,out}^-(n) \leq \max[0, B_i^-(1) + \min_{k \in K}\{\delta_{i,in}^-(n+k) - B_i^+(k+1)\}] \tag{3.50}$$

□

Finally, to predict the minimum output event distances more accurately, it is worthwhile to reconsider the case where the resource can be transiently over-occupied. Theorem 3.11 will in this case often return an overly conservative 0 as the minimum distance between a small number of events. The minimum execution time C_i^- of the processing task τ_i can be used to ameliorate this prediction. If all events from the same event stream are processed in order by the same task, the events at the output will be produced at least with a distance C_i^-. More generally, the busy time $B^-(q)$ delivers the minimum time to finish a set of coinciding activations. This is exploited in the following theorem.

Theorem 3.12. *At the output of a task τ_i the distance between any q produced events is never smaller than*

$$\delta_{i,out}^-(n) = B_i^-(q-1) \tag{3.51}$$

Proof. In order for 2 events to be observable at the output of task τ_i, τ_i must have been activated by two events. Each of these task instances will produce exactly one event during its execution. Both executions can directly succeed each other. Assuming that the events are produced at the end of the task execution, the minimum distance

between the 2 produced events is $\delta^-_{i,out}(2) = B^-_i(1)$. (If events are produced during the execution, the minimum distance would be zero: $\delta^-_{i,out}(2) = 0$.)

Any further events to be observed at the output require another task activation. Every task activation requires at least $B^-_i(1)$ to execute. A third activation can therefore begin its execution no sooner than $B^-_i(1)$ after the previous. Thus, the third event may not be produced $\delta^-_{i,out}(3) = B^-_i(2)$ after the first. This reasoning can be continued for further events. □

Both Theorems 3.11 and 3.12 have been shown to provide conservative minimum distances between any number of events. Consequently, the maximum of both equations delivers the most accurate results.

A method for output jitter calculation using standard event models has also been proposed in [Hen07a]. It can be shown that the method proposed in Theorem 3.11 is a generalization of this concept for arbitrary event models and subsumes its results.

3.6.2 Derivation of Maximum Event Distances

The above calculated minimum distance between events is of major importance to calculate the worst-case load on a particular resource. But for many setups, e.g. in control loops, the best case is of equal importance. For its derivation it is necessary to provide the minimum load, which is given by the maximum distance between events. This can be computed according to the following theorem. Similar to the previous consideration about minimum distances, the key idea is to minimize the finishing time of one event and maximize the finishing time of a successive event.

Theorem 3.13. *Given a task τ_i with a maximum busy time function $B^+_i(n)$ and a minimum busy time $B^-_i(1)$. When the maximum distance between n arriving events is bounded by $\delta^+_{i,in}(n)$ then the maximum distance between n events at the output is bounded by $\delta^+_{i,out}(n)$ as follows:*

$$\delta^+_{i,out}(n) = \max_{k \in K} \{\delta^+_{i,in}(n-k+1) + B^+_i(k)\} - B^-_i(1) \tag{3.52}$$

Proof. The proof of this theorem follows along the lines of Theorem 3.11: The first considered event can not be processed earlier than $B^-_i(1)$ after its arrival. The finishing time of a successive event n is maximized, if it arrives as late as possible. Then let all intermediate events also arrive as late as possible and span the largest possible busy windows, causing the maximum disturbance to event n. The difference between the production of the first and the n-th event is the maximum event distance. □

3.6.3 Monotonicity with Respect to Input Parameters

The output event model computation provided in this section can directly be used to derive the flow of events along a chain of tasks, or other resources. But its application

is also possible in the context of multiprocessor systems with mutual dependencies between task activations and the resulting output and input event models. To address such a setup one can resort to the compositional analysis approach provided in Chapter 2. Its convergence and conservativity relies on the properties of the contributing analysis components that were identified in Corollary 2.8. With respect to the output event model computation, we have already seen that the output event models form a complete partial order in Section 3.3.1. It now remains to be shown that the analysis behaves monotonically with respect to its input parameters.

For convenient notation, assume that $\delta^-_{i,out}(n)$ is the output event model computed with model parameters B_i^+ and $\delta^-_{i,in}$, and $\delta^{-'}_{i,out}(n)$ is the output event model computed with parameters $B_i^{+'}$ and $\delta^{-'}_{i,in}$.

Lemma 3.14. *The output event model computation according to Theorem 3.11 is monotonic with respect to its input parameters.*

$$(\forall u : \delta^{-'}_{i,in}(u) \leq \delta^-_{i,in}(u) \quad \wedge \quad \forall v : B_i^{+'}(v) \geq B_i^+(v)) \tag{3.53}$$

$$\Rightarrow \quad \forall n : \delta^{-'}_{i,out}(n) \leq \delta^-_{i,out}(n) \tag{3.54}$$

(Note that a *decrease* in the minimum distance between incoming events $\delta^-_{i,in}(u)$ translates into an *increased* maximum load on the resource $\eta^+_{i,in}(\Delta t)$.)

Proof. First, recall that Lemma 3.9 provided a set of relevant candidates K that may lead to a worst-case exit time of an event; this set is also used in Theorem 3.11. However, the set does not actually impact the worst-case, it only helps to avoid computing irrelevant candidates. Thus, even though we have a growing number of candidates when $B_i^+(k)$ increases or $\delta^-_{i,in}(k+1)$ decreases, this can not influence the resulting $\delta^-_{i,out}(n)$, and thus its monotonicity.

Moreover, we have assumed in Theorem 3.11 for simplicity, that the best-case behavior $B_i^-(1)$ is not iteratively computed, but constant from the beginning of the analysis (e.g. given by the task's best-case execution time).

Thus, it suffices to show that $\min_{k \in K}\{\delta^-_{i,in}(n+k) - B_i^+(k+1)\}$ in equation (3.41) is monotonic: The lemma's assumptions state that for each $n+k$, that the first term in the minimum function (the minuend) is non-increasing, and for each $k+1$ the second term (the subtrahend) is non-decreasing. Consequently, their difference is for every k non-increasing. Thus, the minimum over all k is non-increasing. □

One input parameter of the output event model computation is the task's multiple event busy time function, which is provided by a dedicated analysis function. For example, a conservative analysis for the multiple event busy time under static priority preemptive scheduling was provided in Lemma 3.3. We now show that its analysis results are also monotonic with respect to its input parameters, namely the event models of all tasks mapped to the task's resource.

Lemma 3.15. *The multiple event busy time according to Theorem 3.3 is monotonic with respect to its input parameters.*

$$\forall j, m : \delta^{-'}_{j,in}(m) \leq \delta^{-}_{j,in}(m) \quad \Rightarrow \quad B_i^{+'}(n) > B_i^{+}(n) \tag{3.55}$$

Proof. We know from equation (3.5) that decreasing the distance between events translates to an increase of the number of events that may be observed in a time window.

$$\delta^{-}_{j,in}(m) \leq \delta^{-'}_{j,in}(m) \quad \Rightarrow \quad \eta_{j,in} j^{+}(\Delta t) \geq \eta^{+'}_{j,in}(\Delta t) \tag{3.56}$$

In Theorem 3.3, the multiple event busy time $B_i^+(q)$ of a given q is the least fixed point *lfp* of equation (3.23). Let $RHS(B_i^+(q))$ denote the right hand side (the workload function) of said equation with parameter $B_i^+(q)$:

$$lfp = \inf\{B_i^+(q) \mid B_i^+(q) = RHS(B_i^+(q))\} \tag{3.57}$$

Due to the stepwise behavior of the η function, and the inclusion of the core execution time of the investigated task in equation (3.23), we have for $B_i^+(n) = 0$:

$$0 < RHS(0) \tag{3.58}$$

i.e. the workload in the critical instant surpasses the provided execution time. Hence the value of RHS is larger than its parameter for any value (time interval) x that is smaller than the fixed-point:

$$\forall x < lfp : RHS(x) > x \tag{3.59}$$

Furthermore, RHS is a concatenation of order-preserving functions (addition and multiplication), and thus, increasing the value of any of its input parameters, will lead to a non-decreasing RHS. Let RHS' be the original RHS increased by an arbitrary value, or the result of any of its input parameters increased by an arbitrary value, such that:

$$\forall x : RHS'(x) > RHS(x) \tag{3.60}$$

Then together with Equation 3.59 follows

$$\forall x < lfp : RHS'(x) > x \tag{3.61}$$

Thus, no fixed point lfp' of RHS' can exist, with $lfp' < lfp$. Consequently, the multiple event busy time behaves monotonically with respect to its input parameters. □

Given any other scheduling policy and corresponding analyses, one has to again show that the analysis results (i.e. predicted behavior) behaves monotonically with respect to the input parameters (i.e. assumed behavior). However, any reasonable analysis can be expected to behave in such a way. The rationale behind this is as follows: Assume a conservative and tight analysis that is subjected to two different sets of input parameters, where one of the sets contains all possible behaviors of the other. Then the analysis of the more generic model must also consider every behavior that is represented in the more constrained model. Thus, in order to remain conservative it can not return analysis results that represent a smaller set of resulting behaviors.

3.7 Experimental Evaluation

In this section, a set of experiments is conducted to show the validity and benefit of the presented approach.

First, consider a simple example system consisting only of two tasks mapped to a static priority preemptive resource. Table 3.7 shows the setup, and Figure 3.7 shows the calculated output event models of the low priority task T2. The black square-tagged curves show the bounds on the event model derived by the classical method of adding the response time jitter (EM_{RT}), while the inner triangle-tagged curves show the result of the procedure proposed in this chapter (EM_{BT}). It can be seen that the new method predicts tighter bounds on the resulting number of events.

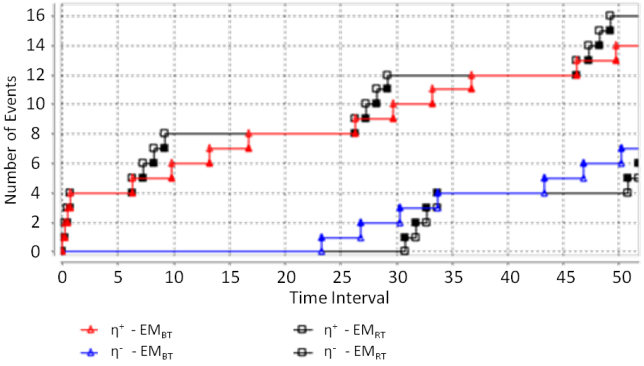

Figure 3.7: Example Output Event Models Propagated [Sch08c].

To quantify the improvement, we rely on three metrics: First, the maximum deviation in the number of predicted events. In the example, the original method calculates the number of events by more than 30% larger than the new method. In a time window slightly smaller than 10 time units, the original analysis predicts 8 events, while only 5 may occur. Second, the maximum deviation of the predicted event distances. In

CPU1					CPU2			
task	event model t_i, b, t_o	cet	priority		task	event model t_i, b, t_o	cet	priority
T1	1,3,20	[1,3]	high	→	T3	derived	[2,5]	high
T2	random	[0.25, 3.5]	low	→	T4	derived	[1,4]	low

Table 3.1: Parameters of Example System.

the example the distance between 8 events was predicted to be at least 9 time units, while the lower bound provided by the new method was actually 17 time units. Thus the original method has overestimated by almost 90%.

These two metrics focused on the worst case deviation, in order to capture the overall accuracy, we also compare the "tightness" of the two event models. For this, we calculate for each the area between the upper and lower event curve. A smaller area means a better prediction. In this example, the improvement in area is around 10%.

The increased precision of the proposed approach is due to the correlation of individual event distances and their respective worst-case latencies. The loss of precision can be expected to increase if more than one resource is involved. Reduced event model accuracy leads to a degradation of successive response-time estimates, eventually causing rejection of systems that are actually schedulable.

To further investigate the average benefit, we model a small two processor system with two parallel processing chains (see Table 3.7, task T1 sends data to T3, task T2 sends data to T4). We generated 1000 sets of random event models with periodic bursts at the input of T2 (Parameter ranges: Inner period $1 \leq t_i \leq 4$, burst size $1 \leq b \leq 10$, outer period $t_o = t_i * (b + 1 + r) \mid 0 \leq r \leq 80$). About 10% of the generated event models caused a resource to be overloaded and were discarded. In all other cases we compared the resulting event models at the output of T4. Total analysis time in our implementation was 2 minutes.

The distribution of the maximum deviation of predicted event distances is shown in Figure 3.8a. In most of the experiments, the original propagation method derived around 30% to 40% larger values, and in the extreme cases even 400%, which shows the benefit of using the new method.

Also the overall tightness of the resulting event models improves significantly: In our experiments, the event model area could on average be reduced by 35%. The amount of improvement is dependent on the load imposed on the processors. In Figure 3.8b the change in event model area is plotted against the average distance between events, which linearly influences the processor load. In systems with large event distances (i.e. low utilization), the simple propagation mechanism already captures the behavior accurately. As the load increases, the new event model propagation provides significantly tighter bounds with less than one quarter of the original area.

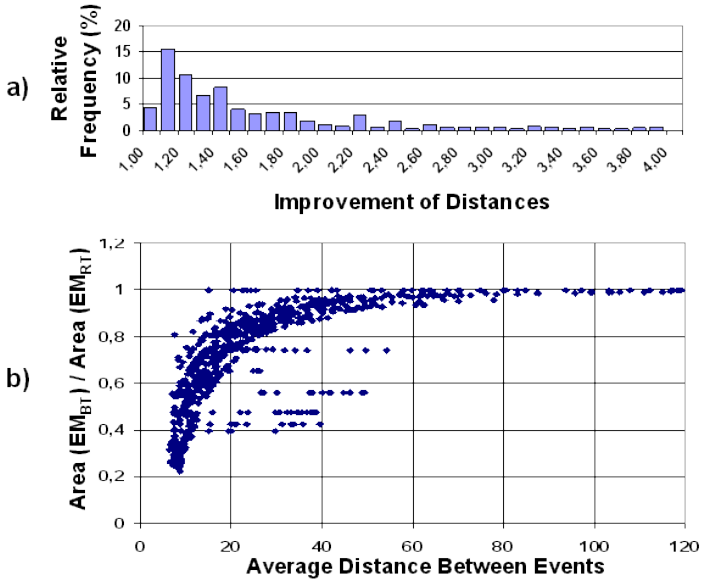

Figure 3.8: Comparing Quality of Obtained Event Models [Sch08c].

3.8 Conclusion

In this chapter a new methodology to investigate the timing of events in a multiprocessor system was introduced. The approach relies on the multiple event busy time metric as an expressive model of the resource service. This metric is more accurate than the classic worst-case response time, yet less complex than the continuous service curve.

A theory was provided that allows to derive the event load and distances of a priori unknown event models within the system. Due to its monotonic properties, it can be embedded into the iterative analysis procedure presented in Chapter 2 to tackle also complex systems with functional cyclic dependencies. The proposed analysis supports arbitrary event model functions and a large variety of different heterogeneous scheduling policies. The experiments have demonstrated the applicability and qualitative improvement over previous methods.

4 Pipelined Path Latency

4.1 Introduction

A common setup in real-time systems is that multiple tasks on the same or different processors are subsequently involved in the processing of an event. For example in automotive systems, multiple controllers are usually involved in a sensor-actor chain. But also in streaming applications such as multimedia, the sequential processing on multiple resources is a common measure to address throughput constraints. Such a processing chain opens opportunities to use specialized components, and benefit from increased throughput through event pipelining. These benefits can however only be exploited for real-time systems if an accurate analysis is available that captures the timing behavior.

The classical approach to derive the end-to-end latency has been to accumulate the individual task worst-case response times along the path [Sun95, Hen05]. This simple summation is then a conservative estimate, but it also leads to a large overestimation in the case of bursty event arrivals: If a burst enters the system this translates into large local worst-case response times, as an event may have to wait for preceding events of the same stream to be finished. Usually, such a burst can occur anywhere along the considered path, and consequently all local worst-case response times will be relatively large. In reality however, an event that has been delayed by its predecessors on one resource can not be fully delayed by the preceding events again on the successive resource. During its waiting time the preceding events have continued to be processed on the successive resources as in a pipeline.

Note that the calculated *local* worst-case response times and traffic estimates may be correct and conservative, only that there is no actual scenario in which one event experiences the worst-case response time on *each* resource along a path. This effect has been called the "pay-burst-only-once" phenomenon [LeB01]. Similar to the approach in network calculus, we avoid this problem by providing a dedicated analysis for the complete path that considers the correlation between event arrival times and local response times in Section 4.2.

Another challenge to the analysis of processing chains is introduced by the possible application topologies. For example, the workload may be processed in parallel by different task instances which may or may not be mapped to the same processor. Such "fork/join" application topologies are very common in multimedia. Compositional analysis approaches that enforce a rigid hierarchy between local and global scope

have great difficulty with capturing the correlation between the events at a "join" node [Jer05, Per08]. The approach provided in this chapter tracks the events on each path to the common predecessor, which provides the necessary synchronization information as shown in Section 4.3.

Finally, cyclic data- or scheduling-dependencies between tasks introduce timing problems such as a reduced achievable throughput and aggravated end-to-end latency. This is well covered by analysis approaches that work on the complete graph (such as [Mor07]), but poses a challenge to compositional analyses that iterate through the components of the application topology. We will differentiate between functional cycles and non-functional cyclic dependencies in our proposed analysis approach in Section 4.4.

The chapter is organized as follows. Next, we will provide a small introductory example. We then review the related work in Section 4.1.2. Section 4.2 then provides the basic analysis concept for simple paths, which we extend to consider fork- and join structures in Section 4.3, and cyclic dependencies in Section 4.4. Section 4.5 then provides an experimental evaluation, and we conclude in Section 4.6.

4.1.1 Example

The rationale behind the improved path latency analysis is explained in the following example.

Consider a system with 2 CPUs and 2 Buses, for which a possible gantt diagram is depicted in Figure 4.1. A critical message needs to be periodically transported via the path $\{C0, T1, C2, T4\}$, and arrives already with a small jitter. On the two CPUs, higher priority tasks $T2$ and $T3$ are periodically activated, with no known correlation to $T1$ and $T4$ (such as offsets).

Once the messages have been transported over $Bus1$, two scenarios are possible that may lead to a worst-case latency of an arbitrary event 0:

1a) The interference by the higher priority task $T1$ is aligned with the arrival of event 0 (indicated by the small triangle), and thus the corresponding activation will experience the worst-case multiple event busy time $B^+_{T2}(1)$.

1b) The interference by $T1$ is aligned with the arrival of the preceding event -1, and thus the corresponding activation is delayed by the unfinished previous activation. In this case both events -1 and 0 have been processed $B^+_{T2}(2)$ after the arrival of event -1.

Task activations further in the past may not interfere in this example due to a sufficient distance between the activating events. Scenario 1b) produces a later production time of event 0 at the output of $CPU1$:

$$lat_{in \to T2} \leq \max[0 + B^+_{C1}(1) + B^+_{T2}(1), \\ 0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(2)] \tag{4.1}$$

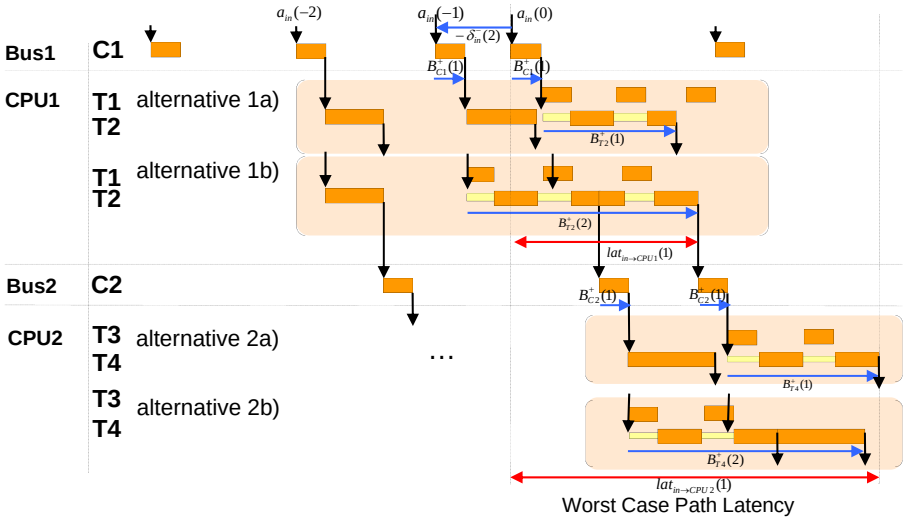

Figure 4.1: Different Candidates For Worst-Case Path Latency of Event 0 being processed along the path $\{C0, T1, C2, T4\}$.

This reasoning can be continued for the subsequent resources. On $Bus2$, the additional latency of event 0 is straight-forwardly bounded by $B^+_{C2}(1)$. There can be no interference from preceding activations, and thus no scenarios need to be checked.

On the next component, $CPU2$, there are again two relevant scenarios which may cause the largest latency increment for event 0 to be processed by task $T4$. Event 0's finishing time is maximized, if the interference of task $T3$ is aligned with the arrival of event 0, as depicted in Scenario 2a. Aligning the interference with the arrival of the previous event, as shown in Scenario 2b, can in this case not lead to a larger production time of event 0 at the output of $T4$.

The worst-case path latency for event 0 is in this example given by equation (4.2), in which all relevant scenarios are listed. Only two backlogged events may interfere on $CPU1$ or $CPU2$, because the arrival of three events is always further apart in time than the time to process two events on any resource.

$$\begin{aligned} lat_{in \to T3} \leq \max[&0 + B^+_{C1}(1) + B^+_{T2}(1) + B^+_{C2}(1) + B^+_{T4}(1), \\ &0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(2) + B^+_{C2}(1) + B^+_{T4}(1), \\ &0 - \delta_{in}(2) + B^+_{C1}(1) + B^+_{T2}(1) + B^+_{C2}(1) + B^+_{T4}(2)] \end{aligned} \tag{4.2}$$

The concept of the worst-case path latency can be generalized to the worst-case processing time for a number of events. The classical latency is then a special case

of this n-event latency for $n = 1$. This metric can be useful, for example, when a series of samples needs to be collected before a valid output can be calculated. Also, it allows to consider multiple events in larger time frames, which allows to contain the influence of transient effects.

4.1.2 Related Work

All approaches for the performance analysis of multiprocessor systems bring a method to compute the end-to-end latency of events that are processed by sequential tasks on multiple resources. The simplest way to conservatively determine the latency is by accumulating the local worst-case response times as is done in e.g. [Sun95] and [Hen05]. However, this procedure is inaccurate in the case of bursty event occurrence owing to the 'pay-bursts-only-once" problem. A burst of events can in general occur at the input of any task along a path — leading to large local worst case response times — but the same event processed along the path can not experience this delay at each task.

System modifications such as traffic shaping have been proposed to avoid the problem altogether and increase the time predictability of a system [Bet92] [Wan06b], but this reduces the system's flexibility and may not always be feasible due to necessary hardware features.

Better estimates in the presence of dynamic event flows can be achieved through the convolution of component behavior along the path as is done in [Cru91a, LeB01] and [Cha03a]. However, these methods rely on the concept of continuous time service curves. In general the folding operations can therefore be computationally intensive. The approaches have not been extended to cover analysis dependencies of functional cycles. The proposition of Section 4.2 is to perform similar folding operations in discrete time domain using the multiple event busy time. This naturally limits the computed values to the critical candidates.

In [Sti98] the timing behavior of processing elements is abstracted with latency-rate servers. Such servers deliver a latency and rate guarantee to all incoming traffic arriving above a certain rate. The observation is that a chain of such servers can be reduced to a substitute server, leading to an efficient analysis. The topology is however purely sequential, and the provided service has to fit the latency-rate model, finer service models (such as provided by priority-based scheduling) have to be conservatively approximated. A similar substitute concept was followed in [Jay07] for processors scheduled with fixed-priority scheduling. However, the approach is specifically aimed at homogeneously scheduled systems, and does not allow for any cyclic functional or non-functional dependencies.

A different approach was taken in [Mor07] by modeling the system as a dataflow graph (including any a priori boundable scheduling delays as suggested in [Wig07]). In single-rate dataflow graphs, the adherence to latency constraints can be checked by adding a reverse edge from the path end. The introduced cycle translates the latency

constraint to a throughput constraint that can be checked with traditional maximum cycle mean algorithms (as in [Das04]). Furthermore a procedure to compute the latency is given for periodic, sporadic and bursty sources and arbitrary application topologies. However, the application is accurate only when applied to the said stimulation patterns and scheduling policies. The observation in [Mor07] and also in [Thi09] that only a bounded event history has an effect on specific event occurrences is also supported by our findings (see Section 4.4). Of course, timed automata can also be used to derive end-to-end path latencies [Moh08] with high accuracy, but subject to the known limitations with respect to scalability.

In [Jer05] and [Hai07] the activation delay of multiple inputs in a "join"-task is provided as the maximum possible deviation between the arriving events as bounded by their event streams. But if the incoming streams have a common source, the delay is usually far smaller due to the related event arrival times. This is considered in the latency computation of [Hua07], where the activation delay is bounded the difference in the latency along the sub-paths. The analysis provided in this chapter also captures this effect (see Section 4.3).

A highly interesting aspect that is not considered in this chapter is the case where tasks do not communicate via FIFO buffers, but data is exchanged via registers (which is often the case e.g. in automotive). In this case data may be lost or duplicated due activation jitter or over- and undersampling. In this case, different latency "semantics" have to be distinguished (such as latest time of reading the data as opposed to the maximum age of the data). These implications have been highlighted in [Fei08].

4.2 Recursive Path Latency Computation

4.2.1 Definitions

We assume that each task is *activated* once at the moment when one event has arrived at each of its input ports ("AND activation semantic"), it is then *running* (i.e. *ready* or *waiting*) until it has been assigned sufficient time on the processor and its resources and then *terminates* (this model will be refined in Chapter 6). Each task instance produces exactly one event at each of its output ports sometime before it terminates. Corresponding to the notation in Chapter 3, all events are numbered according to the sequence of their occurrence — events occurring later receive higher numbers. We will later focus on an arbitrary event 0, thus preceding events can have negative numbers.

Definition 4.1 (Numbering)**.** *A task is activated for the n-th time when it has received n events at each of its inputs. Each task instance produces exactly one event with the same number at each of its outputs.*

The arrival time of an event n that leads to the activation of task τ_i is denoted with $a_i(n)$. The time at which the resulting task instance produces its event is denoted

by $e_i(n)$. Tasks process the events of an event stream in-order. This is a typical assumption in scheduling theory matching the design practice. Prioritized events are modeled with separate event streams.

Tasks in the system can be chained, such that the event produced by one task leads to the activation of another. For each task τ_i in the system, let $S(\tau_i)$ be the set of τ_i's direct successors, i.e. those tasks that require an event from τ_i in order to be activated; and let $P(\tau_i)$ be the set of τ_i's direct predecessors, i.e. those task's from which τ_i requires an event in order to be activated. Thus, we have $\tau_i \in S(\tau_j) \Leftrightarrow \tau_j \in P(\tau_i)$. A sequence of tasks in which each element activates the next is called a *path*:

Definition 4.2. *A* path *is an ordered set of tasks* $\mathbb{P} = \{\tau_1, \ldots, \tau_{end}\}$*, in which for each two neighboring elements* $\tau_i, \tau_j \in \mathbb{P}$ *we have* $\tau_i \in S(\tau_j)$ *(and* $\tau_j \in P(\tau_i)$*).*

Deriving the end-to-end latency of events that are processed along a path is the primary concern of this chapter. To track the processing of events along a path, we introduce need the concept of *causal dependence*, which provides the relationship between events at the input of a task and those produced by it.

Definition 4.3 (Causal Dependence of Events)**.** *An event b is* causally dependent *on event a, if b is produced by the same task instance that consumes a, or any successive instance of the same task. All task instances that are activated by causally dependent events are causally dependent. All events produced by causally dependent task instances are causally dependent (transitively).*

With these definitions, we can formally define the path latency as illustrated in the above example.

Definition 4.4 (Path Latency)**.** *The n-event end-to-end latency of path* $\mathbb{P} = \{\tau_1, \ldots, \tau_{end}\}$ *is the maximum time distance between the arrival of an arbitrary event* 0 *at the input of* τ_1 *and the production of the n-th causally dependent event at the output of the last task* τ_{end}*.*

$$lat_{\mathbb{P}}(n) = \max \quad e_{end}(n-1) - a_1(0) \tag{4.3}$$

4.2.2 Computing the Path Latency

The maximum value of $lat_{\mathbb{P}}(n)$ depends on the actual timing of the events that are processed along the path as well as the timing of all other events in the system. In our approach, we abstract the timing of the other events with the help of the multiple event busy time model per involved task, which represents the local worst-case behavior. These local results are then composed in order to derive a maximum value for $lat_{\mathbb{P}}(n)$, taking into account the inherent pipelining of event processing along the path.

Let us first focus on the timing of *simple* paths, in which each task along the path only has one predecessor, and thus its activation does not require events that are

produced by tasks that are not in the path. We will drop this focus in Section 4.3, but for now, the activation time of a task τ_i is exactly the production time of the event from its predecessor τ_j:

$$a_i(n) = e_j(n), \text{if } \tau_i \in S(\tau_j) \tag{4.4}$$

Furthermore, we know that events arriving at the first task of the path belong to the same event stream, and thus their arrival times are correlated. Their minimum (and maximum) distances are given by their comprising event model as provided by Definition 3.3 and bounded by Lemma 3.8.

Using our knowledge about the maximum exit times of events (Corollary 3.10) together with the observation in equation (4.4), we now can derive the latest possible times for the occurrence of events along a simple path through recursion. The following theorem bounds the path latency as defined above.

Theorem 4.1. *Let $\{\tau_1, \tau_2, \ldots, \tau_{end}\}$ be the (ordered) set of tasks along a simple path τ_j. Then the n-event latency of $\mathbb{P}$ can be recursively computed as follows:*

$$lat_{\mathbb{P}}(n) = e_{end}(n-1) \tag{4.5}$$

$$e_i(n) \leq \max_{k \in K}\{a_i(n-k) + B_i^+(k+1)\} \tag{4.6}$$

$$a_i(n) = e_j(n), \text{ if } \tau_i \in S(\tau_j) \tag{4.7}$$

$$a_1(n) \leq \begin{cases} -\delta_{1,in}^-(-n+1), & \text{if } n < 0 \\ \delta_{1,in}^+(n+1), & \text{if } n > 0 \end{cases} \tag{4.8}$$

Proof. We know from Corollary 3.10 that the production time of any event n produced by a task τ_i along the path is bounded by

$$e_i(n) \leq \max_{k \in K}\{a_i(n-k) + B_i^+(k+1)\} \tag{4.9}$$

with the bounded set of relevant events K (Lemma 3.9). Furthermore, we know that along a simple path every task is activated by the events produced by its predecessor, and thus

$$a_i(n) = e_j(n) \text{ if } \tau_i \in S(\tau_j) \tag{4.10}$$

Finally, the first task in the path has no predecessor, but the events at its input are constrained by the specified event model. This allows us to bound the latest time at which events may enter the path in relation to each other according to Lemma 3.8:

$$a_1(n) \leq \begin{cases} a_1(0) - \delta_{1,in}^-(-n+1), & \text{if } n < 0 \\ a_1(0) + \delta_{1,in}^+(n+1), & \text{if } n > 0 \end{cases} \tag{4.11}$$

Thus, in effect, via the simple translation in (4.10), equation (4.9) can be expanded: every $e_i(n)$ is transitively a function of $a_1(n)$ and via (4.11) a function of $a_1(0)$. Because the maximum operation is distributive over the addition, we can isolate $a_1(0)$ from each term in (4.11) and in the expanded versions of equation (4.9). We can bound $e_{end}(n-1)$ from Definition 4.4 with respect to the arrival time $a_1(0)$ of event 0.

$$lat(n) \quad \leq \quad e_{end}(n-1) - a_1(0) \tag{4.12}$$

The isolated $a_1(0)$ cancels with $-a_1(0)$. Thus, (4.5) and the theorem follow.

□

The minimum end-to-end latency can be calculated with a similar method. However, for most realistic cases the minimum latency is simply the sum of the best case response times — it is therefore not explored further in this thesis.

4.2.3 Reconciling Worst-Case Latency with Long-Term Throughput

The reasoning about a system's performance is often reduced to very simple metrics. In the control domain, the worst-case latency is of utmost importance to ensure control stability and timely reaction. In transformative applications, such as multimedia, the long-term sustainable throughput is more important. These reduced concepts are not adequate to address setups in which the timely processing of a finite amount of workload has to be ensured: The long-term throughput provides no indication about the service provided in bounded time intervals or to finite bursts of data, and the worst-case latency drastically overemphasizes the critical system conditions (such as interfering workload).

This situation is ameliorated by Theorem 4.1, which allows computing the worst-case path latency for $n = 1$ events, as well as the maximum amount of time between the arrival of an event at the path beginning, and the production of the n-th causally dependent event at the path end. Thus the metric delivers a more fine-grain indicator of the achievable performance.

For example, let the workload consist of n events. This could be an image that consists of n packets that are transmitted and processed via a chain of tasks. We know from Theorem 4.1 the maximum amount of time until the last of the n events has been processed by the system. Usually, this latency is far smaller than n times the worst-case latency ($lat(n) < n \cdot lat(1)$), due to the pipelined processing and the rareness of transient overload situations. Thus, we are able to provide a service guarantee — without overemphasizing the worst-case. Furthermore, the actual long-term throughput can be conservatively approximated from the n-event path latency. If we know that the worst case latency along the path is $lat(1)$ than we know that the system can always sustain a throughput of at least $T_{\mathbb{P}} \geq 1/lat(1)$. If we know that the total time to process two events along a path is $lat(2)$ than the system can always

sustain a throughput of at least $T_{\mathbb{P}} \geq 2/lat(2)$, which is a slightly more accurate bound on the long-term throughput. This reasoning converges to the following lower bound approximation sustainable for the long-term throughput for sufficiently long streams of data:

$$T_{\mathbb{P}} = \lim_{n \to \infty} \frac{n}{lat_{\mathbb{P}}(n)} \tag{4.13}$$

This reasoning remains valid also in the presence of the following extensions that allow deriving the path latency for more complex application topologies.

4.3 Fork and Join Application Topologies

Typical applications in embedded systems consist of more than just a sequence of tasks that are sequentially activated on the arrival of new data. In an automotive application for example, data may be needed from different sensors before computation can begin, and the computed results may be distributed to multiple actuators afterwards. In multimedia, intermediate parallelization is very common, for example, when the frame of a video stream is split into a number of subframes that are then processed in parallel on an array of hardware elements, before the result is again merged for further processing. Such applications commonly feature *join* and *fork* structures.

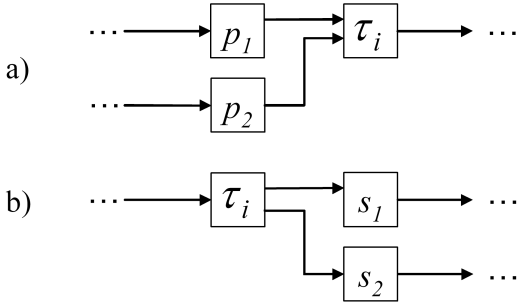

Figure 4.2: Possible application topologies: a) join structure , b) fork structure

To address these application structures, the computation of the event exit times from Lemma 3.7 can be extended. We have defined that a task a) is activated for the n-th time no later then when n events have arrived at each of its inputs and b) produces the n-th event on each of its outputs no later than when its n-th activation is finished.

4.3.1 Multiple Outputs

Firstly, a task with multiple outputs will produce one event on each output when it has finished execution.

Lemma 4.2 (Fork)**.** *Let a task τ_i have multiple outputs and let $S(\tau_i)$ be the set of successor tasks that are connected to these outputs via edges $i \to s$. If τ_i finishes the activation n at time $e_i(n)$ then event n arrives at the connected input of each of the successor tasks no later than*

$$\forall s \in S(\tau_i) : a_{i \to s}(n) = e_i(n) \tag{4.14}$$

Proof. Follows directly from the definition of the task behavior. □

4.3.2 Multiple Inputs

A task may also process multiple streams of incoming data. This raises the question about which arrival of data will lead to the tasks activation. Different semantics have been proposed to model this behavior: The most widespread use is given by the "AND" activation semantic (e.g. in [Jer05]), which activates the task when an equal number of events has arrived on each of its inputs. Such a behavior will be shown for example by a task that reads data from multiple sensors, or joins the results of multiple threads.

Other activation semantics are also possible; for example, the "OR" semantic, which leads to a task activation whenever an event arrives on either input, or the "triggered" activation that only reacts to events at a specific input. The timing implications of these semantics have been investigated e.g. in [Jer05, Hai07, Rox08]. In the present thesis we assume "AND" activation semantics, and use its synchronizing behavior to maintain a consistent numbering of events and accurately trace the events along diverging paths. The following lemma bounds the event production of a task τ_i that has multiple inputs. For this an intermediate variable $a'_i(n)$ is introduced, which represents the time at which task τ_i has received sufficient events from its predecessors to begin its n-th activation.

Lemma 4.3 (Join)**.** *Let task τ_i have multiple inputs, and $P(\tau_i)$ be the set of the direct predecessor tasks of τ_i at each of its inputs, where events arrive no later than $a_{p \to i}(n)$ from predecessor $p \in P(\tau_i)$, then the event n is produced by task τ_i no later than*

$$e_i(n) \quad \leq \quad \max_{k \geq 0} \{a'_i(n) + B_i^+(k+1)\} \tag{4.15}$$

with

$$a'_i(n) \quad \leq \quad \max_{p \in P(\tau_i)} [a_{p \to i}(n)] \tag{4.16}$$

Proof. The task will be ready to execute its n-th activation when it has received an equal number of n events from each of its direct predecessors. This will be the case no later than $a'_i(n)$. Once an activation is ready, it will be processed after $B_i^+(k+1)$, depending on the amount k of unfinished preceding activations. The reasoning now follows the proof of Lemma 3.7. □

Lemma 4.3 can now be used to extend the recursive path latency computation in Theorem 4.1, by replacing equation (4.7) with (4.15) and (4.16).

4.3.2.1 Synchronized Inputs with a common Predecessor

There are now two interesting cases to consider: Either the incoming edges have a common predecessor task, in which case the respective event timing is related, or the events at the inputs arrive from unrelated sources, in which case the timing of the incoming events is determined by the amount of "asynchronicity" between the sources.

The case in which the inputs of τ_i have a common root in a task τ_r is directly considered in Lemma 4.3 together with the recursive application according to Theorem 4.1 and Lemma 4.2. The task τ_r will produce its outputs no later than given by a time $e_r(n)$. The events at each output of τ_r will then traverse through different "sub-paths" before they arrive at the different inputs of τ_i. As no events are lost or generated along the sub-paths, the numbering remains valid and the maximum operation in (4.16) identifies the sub-path with the longest time since $e_r(n)$. This accurately captures the latest possible activation time of τ_i.

4.3.2.2 Non-Synchronized Inputs

Some of the inputs of τ_i may also have independent roots, i.e. when data from different sensors is required for an activation, but that data is sent without further synchronization. The derivation of the worst-case arrival times of the incoming events according to Theorem 4.1 then ultimately requires the token arrival times of tasks that have no further predecessors (sources). One of these tasks without predecessors will be τ_1, i.e. the beginning of the investigated path $\mathbb{P} = \{\tau_1, \ldots, \tau_{end}\}$, for which $a_1(n)$ is defined in Theorem 4.1. Other tasks without predecessors, denoted as *path-external sources* may be completely unrelated. For each path-external source, the timing of the events it produces is described by their comprising event model, but the timing relation to the events entering the path at $a_1(n)$ is open.

Each of these sources must essentially have the same long-term throughput to avoid buffer overflow at one of τ_i's inputs. Still, the different sources may produce data with different patterns or jitter, which can cause an activation delay and increase the latency along path $\mathbb{P}$. The magnitude of this delay is given by the distance between the arrival of an event along path $\mathbb{P}$ and the arrival of the corresponding event on any of the other inputs. This metric, which we call the maximum *drift* between asynchronous inputs, has also been studied in [Wan06a].

Let $P^{ext}(\tau_i)$ be the set of direct predecessors of a task τ_i that are not causally dependent on a task in path $\mathbb{P}$. Then the delay between the activation of task τ_i due to such external events is bounded by the following drift between the arrival of events:

$$D_i \leq \max_{n \geq 2} \max_{p \in P^{ext}(\tau_i)} [\delta^+_{p,in}(n) - \delta^-_{i,in}(n)] \tag{4.17}$$

where $\delta^+_{p,in}(n)$ is the maximum distance between events arriving from the external predecessors p, and $\delta^-_{i,in}(n)$ is the minimum distance between events arriving from the predecessor along the path $\mathbb{P}$. With this, we can rephrase (4.16), so that it becomes independent from the timing of external sources.

$$a'_i(n) \quad \leq \quad \max_{p \in P_i \setminus P^{ext}(\tau_i)} [a_{p \to i}(n)] + D_i \tag{4.18}$$

Computing (4.17) can be computationally elaborate if performed in general (see also [Wan06a]). But depending on the parameters of the underlying event model, this can be accomplished more efficiently. In [Jer05] the delay D_i was analytically derived for the case where the event models are represented as standard event models.

4.4 Cyclic Dependencies

In a general multiprocessor setup, functional cycles and non-functional cyclic dependencies may exist between different tasks. These disrupt a straight-forward analysis procedure as proposed in the previous section. However, our goal is to admit both types of cyclic dependencies in order to increase the scope of systems that can be investigated.

Our solution is two-fold: The non-functional dependencies are resolved through the iterative analysis presented in Chapter 2, which has the task activating event models and task's multiple event busy times as a result. Functional cycles are tackled through a stop condition that breaks an infinite recursion.

4.4.1 Non-functional Cyclic Dependencies

Non-functional dependencies are the implicit result of resource sharing in a system. Figure 4.3a) for example shows a system with two tasks mapped to different processors. Assume a priority driven scheduling and let τ_4 and τ_2 have the highest priorities on their respective resources. Then the response time of τ_1 can not be computed without knowledge of τ_3's output event model, and τ_3's output event model can not be computed without an analysis of τ_1. Such cyclic dependencies are common in larger setups with multiple processors and buses. Often they can not be avoided by design, in order to comply with legacy implementations and fulfill all timing constraints.

Such dependencies are tackled through the iterative analysis procedure that we have presented in Chapter 2, a concept that has already been used successfully in many setups [Ric02b, Cha03a, Jon08, Ste08]. When the analysis has converged, task activating event models δ^-_i, δ^+_i are known for every task in the system, whether it is time-triggered or event driven. These event models are then the basis for computing for each task the worst-case response time R_i^{max} and multiple event busy time function B^+_i. This enables the computation of the path latencies according to Theorem 4.1.

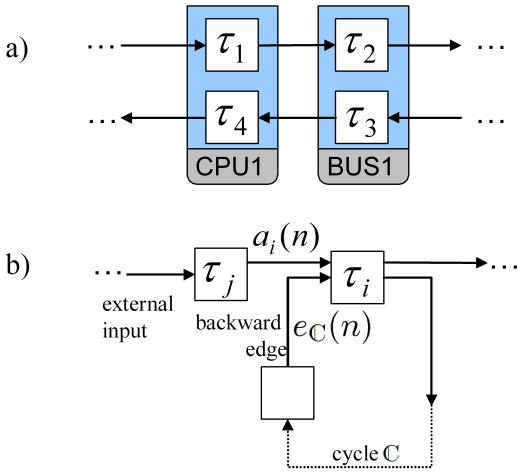

Figure 4.3: Cyclic Dependencies: a) non-functional cyclic dependency, b) functional cycle

4.4.2 Functional Cycles

Functional dependencies are inherent to the application structure, and independent of the hardware mapping. A typical example of a functional cyclic dependency is a control application that consists of a loop that contains a sampling operation (*sensor*), the computation of a new parameter (*controller*), and the controlling actuators (*plant*). Here, sufficient throughput is required for control stability. Functional cycles are also common in signal processing applications, where the result of a preceding activation is reused as an input for the current filtering step.

In systems with such cycles, the recursive application of Theorem 4.1 and Lemma 4.3 will cause an infinite recursion, because an activation of a task τ_i in a cycle may eventually depend on the finishing time of a preceding activation of the same task. This problem will now be addressed by bounding the necessary number of recursions. The main idea is that in a *schedulable* system, the processing of an event in the cycle can not be delayed by events that have arrived sufficiently far in the past.

Formally, a functional cycle $\mathbb{C}$ is a sequence of tasks with cyclic activation dependencies. Transitively, the activation of every task $\tau_j \in \mathbb{C}$ is causally dependent on a preceding activation of the same task. Some consecutive tasks in $\mathbb{C}$ may also lie along an investigated path $\mathbb{P}$. We assume that each cycle has one cycle-external input edge that determines the arrival of new data. Multiple cycle-external inputs can be combined through a merge-task according to Lemma 4.3. A task can be part of

multiple cycles.

For each such cycle, we can identify a task τ_i that has a cycle-external input edge and an input edge that comes from the preceding task in $\mathbb{C}$. Task τ_i then requires events on both the external input and the "backward edge" in order to be activated. If this happens, we say that the event n is *admitted* into the cycle. Given τ_i is a part of the path $\mathbb{P}$, then the cycle-external input will be the one that comes from its predecessor along the sequence of tasks in the path, where we denote the arrival of events with $a_i(n)$.

The dependency constraint imposed by the availability of resources in a cycle can be modeled with *tokens* which limit the number of external events that have been admitted into the cycle but for which the corresponding events (*tokens*) have not returned to the backward edge. For a given cycle $\mathbb{C}$, there are a number of tokens which either correspond to events generated by preceding activations, or represent initial placement of data that is required to admit the first several events. We know from [Bac92] that the number of tokens in a cycle remains constant during execution, as long as each activation of a task on the cycle will consume exactly one token on its input and produce one token on its output edge. Let $d_{\mathbb{C}}$ be the number of such tokens for cycle $\mathbb{C}$, then the n-th activation of every task $\tau_j \in \mathbb{C}$ is causally dependent on the $n - d_{\mathbb{C}}$-th activation of τ_j.

Lemma 4.3 provides an upper bound on the time that the n-th activation of task τ_i is finished, which is when τ_i has received events on all its inputs, including the backward edges of the cycles cycles(i) of which it is a part. For each cycle $\mathbb{C} \in$ cycles(i), let $e_{\mathbb{C}}(n - d_{\mathbb{C}})$ be the "return time" at which the token corresponding to the activation $n - d_{\mathbb{C}}$ is returned to its backward edge. With this, an event n is admitted into the cycle at time $a'_i(n)$ which is no later than

$$a'_i(n) \leq \max[a_i(n), \max_{\mathbb{C} \in \text{cycles}(i)} e_{\mathbb{C}}(n - d_{\mathbb{C}})] \tag{4.19}$$

The return time $e_{\mathbb{C}}(n - d_{\mathbb{C}})$ is a function of the time that task τ_i was started for the $n - d_{\mathbb{C}}$-th time and the time that was necessary to process token $(n - d_{\mathbb{C}})$ by tasks on the cycle. In order to bound the latter, we first establish an upper bound on the amount of time a token may be processed in a cycle.

Lemma 4.4 (Response-Time Bound in Cycles)**.** *In a cycle $\mathbb{C}$ with a total number of $d_{\mathbb{C}}$ tokens, the response time of an arbitrary task $\tau_j \in \mathbb{C}$ is bounded by $B_j(d_{\mathbb{C}})$.*

Proof. Each activation of τ_j consumes one token from its input edge and produces one on its output edge sometime later before it terminates. Thus the number of tokens on the cycle is constant $d_{\mathbb{C}}$, and thus there may be no more than $d_{\mathbb{C}}$ simultaneous instances of task τ_j at any time. Furthermore, none of these task instances is influenced by successive activations due to in-order processing of the activations. Therefore, the

maximum time for any task instance to finish is bounded by the multiple event busy time of $d_{\mathbb{C}}$ events $B_j(d_{\mathbb{C}})$. □

Obviously, if a task τ_j is part of multiple cycles, its response time can not be larger than $B_j(d_j)$, where d_j is the smallest number of tokens $d_{\mathbb{C}}$ in all cycles $\mathbb{C} \in \text{cycles}(j)$ that τ_j is a part of. Lemma 4.4 can now directly be used to bound the round-trip time of tokens in a cycle, i.e. the latest possible time after the admission of an event n into the cycle that it has taken the corresponding token to be returned to the backward edge.

Lemma 4.5 (Round-Trip in Cycles)**.** *In a cycle $\mathbb{C}$ with a total number of d_c tokens, any token has traversed the cycle no later than $\sum_{\tau_j \in \mathbb{C}} B_j(d_{\mathbb{C}})$:*

$$e_{\mathbb{C}}(n) \quad \leq \quad a'_i(n) + \sum_{\tau_j \in \mathbb{C}} B_j(d_{\mathbb{C}}) \tag{4.20}$$

Proof. Follows directly from Lemma 4.4. □

To derive the time $a'_i(n)$ at which event n is admitted into the cycle, we need to check in equation (4.19) whether it is influenced by preceding events that have not been processed by the cycle. With the help of Lemma 4.5, we can establish that the processing of previous tokens on a cycle $\mathbb{C}$ does not impose a constraint on the n-th activation of task τ_i, when

$$a_i(n) \geq a'_i(n - d_{\mathbb{C}}) + \sum_{\tau_j \in \mathbb{C}} B_j(d_{\mathbb{C}}) \tag{4.21}$$

In order to find out whether condition (4.21) is indeed fulfilled for an event n, we need to check further into the past, whether the admission of event $(n - d_{\mathbb{C}})$ into the cycle was itself delayed by preceding tokens, which again may be delayed by their predecessors $(n - qd_{\mathbb{C}})$.

The constraint to break this recursion relies on the fact that the minimum distance to the arrival of previous events grows super-additively, while the possible delay that these past tokens may impose is linearly bounded. Eventually thus, the distance to the arrival of previous events will be sufficiently large to rule out any interference with the present activation. This concept is formalized in the following theorem which bounds the number of successive events that may not be immediately admitted into a cycle.

Lemma 4.6 (Admittance of Preceding Events)**.** *Let a task τ_i be part of a cycle $\mathbb{C}$ and have a cycle-external input at which events arrive at times $a_i(n)$ with a minimum distance of $\delta^-_{i,in}(m)$ between any m arriving events. Then:*

If there is a $k \geq 1$ that fulfills the following inequation

$$\delta^-_{i,in}(kd_{\mathbb{C}} + 1) \quad > \quad k \sum_{j \in \mathbb{C}} B_j(d_{\mathbb{C}}) \tag{4.22}$$

there must be for any event n at least one preceding event j with $j = n - qd_{\mathbb{C}}, 1 \leq q \leq k$ that was immediately admitted:

$$\exists j, j = n - qd_{\mathbb{C}}, 1 \leq q \leq k : a'_i(j) = a_i(j) \tag{4.23}$$

Proof. The proof is by contradiction. Assume that all preceding events $n - qd_{\mathbb{C}}, 1 \leq q \leq k$ arrive at the cycle-external input in such a way that the corresponding tokens on the cycle have not been returned to the backward edge, and thus no incoming event is immediately admitted:

$$\forall q, 1 \leq q \leq k : a_i(n - qd_{\mathbb{C}}) < e_{\mathbb{C}}(n - (q+1)d_{\mathbb{C}}) \tag{4.24}$$

Let the arbitrary time between the arrival and the admittance of the first of the events considered in the theorem be denoted with D and thus:

$$a'_i(n - kd_{\mathbb{C}}) \quad = \quad a_i(n - kd_{\mathbb{C}}) + D \tag{4.25}$$

By iterative application of (4.19) and (4.20) under the assumption (4.24), we can deduce

$$a'_i(n) \quad \leq \quad a'_i(n - kd_{\mathbb{C}}) + k \sum_{\tau_j \in \mathbb{C}} B_j(d_{\mathbb{C}}) \tag{4.26}$$

and with (4.25)

$$a'_i(n) \quad \leq \quad a_i(n - kd_{\mathbb{C}}) + k \sum_{\tau_j \in \mathbb{C}} B_j(d_{\mathbb{C}}) + D \tag{4.27}$$

Also, we know from the properties of the incoming event stream that events arrive at the input with a certain minimum distance according to (3.31), and thus

$$a_i(n) \quad \geq \quad a_i(n - kd_{\mathbb{C}}) + \delta^-_{i,in}(kd_{\mathbb{C}} + 1) \tag{4.28}$$

and because of $a_i(n) \leq a'_i(n), \forall n$:

$$a'_i(n) \quad \geq \quad a_i(n - kd_{\mathbb{C}}) + \delta^-_{i,in}(kd_{\mathbb{C}} + 1) \tag{4.29}$$

Equations (4.27) and (4.29) imply

$$\delta^-_{i,in}(kd_{\mathbb{C}} + 1) \quad \leq \quad k \sum_{\tau_j \in \mathbb{C}} B_j(d_{\mathbb{C}}) + D \tag{4.30}$$

In order to fulfill the condition of the theorem (equation (4.22)), D has to be negative, which is impossible because of (4.25) and $a_i(n) \leq a'_i(n), \forall n$. Thus, we have to conclude that the assumption (4.24) is not fulfilled for at least one q and the theorem follows. □

Theorem 4.6 can now be used to break the infinite recursion in the analysis of the admittance time $a'_i(n)$ of an event n by assuming

$$a'_i(n - kd_{\mathbb{C}}) = a_i(n - kd_{\mathbb{C}}) \tag{4.31}$$

where k is the smallest integer that fulfills condition 4.22. From this, the admittance times of the successive events follow. If no such k exists, the cycle can not be deemed schedulable, because the distance between incoming events could then be steadily smaller than what can be processed by the involved tasks in the cycle.

The smallest k that fulfils (4.22) can be computed numerically for any event model, or based on the parameters of the specific event model. For example, let the events that arrive at the cycle-external input be constrained by a standard event model with parameters period $P_{i,in}$ and jitter $J_{i,in}$ as in [Ric02a]. Then (4.22) can be solved for k and is upper bounded by the following closed-form inequation:

$$k \leq \frac{P_{i,in}d_{\mathbb{C}} - J_{i,in}}{\sum_{j\in\mathbb{C}} B_j(d_{\mathbb{C}}) - P_{i,in}d_{\mathbb{C}}} \tag{4.32}$$

With the computable bound on the activation and finishing times of any task in the system, we have provided a versatile method to compute the maximum end-to-end latency — considering topological hurdles such as fork and join constructs, non-functional dependencies, and functional cycles.

4.5 Experimental Evaluation

We have conducted a set of experiments to show the validity and precision of the presented approach. Consider the path $\{S0, S1\}$ in the example of Figure 4.4 in which events are processed along a diverging path via 6 tasks on 2 buses and 3 CPUs. Each task may be disturbed by higher priority events from another application. The higher priority application is activated by source $S2$ and requires data from a previous iteration to be available. External events arrive at $S2$ with an average distance (period) of 100 and a jitter of 200, and at $S0$ with a period of 20.

The results of two parameter scenarios are plotted in Figure 4.5: In one scenario, all execution times are constantly 5 ("Constant ET"), in the other scenario all execution times along path $\{S0, S1\}$ are variable between 1 and 5 ("Variable ET"). In either case, the *response* times are not constant, because of the dynamic interference from the higher priority application.

The latency calculation that is based on the accumulation of local worst cases ("Add WCRTs") draws the expected overestimation from the pay-bursts-only-once problem. The effect becomes more substantial with growing timing uncertainty (i.e. increasing input jitter at source S0, or introduction of variable execution times). The proposed analysis ("Pipelined") tackles this problem through correlating the local worst case busy times. It is also relatively insensitive to the increase of dynamism. In the given

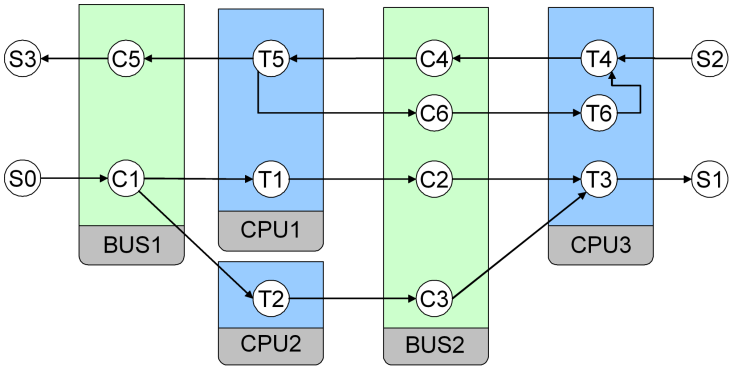

Figure 4.4: Example System.

parameter range the proposed method calculates a between 59% and 79% better end-to-end latencies.

Further experiments of small systems have been performed in the scope of the comparative paper [Per08], in which real-time calculus [Cha03a], SymTA/S [Ric02b], and other approaches were compared. The path latency computation of this chapter was included in the experiments. In one experiment a short chain of 3 tasks on 2 CPUs is investigated. A particular challenge to the analysis is the correlation of events activating T3 and those activating T1. It can be seen in Figure 4.6 that the new path latency calculation "SymTA/S pipelined" is better than the simple additive calculation (SymTA add"). Of all approaches under comparison, our approach is in this experiment closest to the actual worst cases derived with model checking ("Uppaal"), often matching it accurately. This was owed to the fact that the analysis on CPU1 better considers the offsets between the activations of T2 and T3 according to e.g. [Pal98]. [Per08] also contains an evaluation of analysis times — besides its accuracy, the presented analysis is also very fast.

The number of operations that will be performed to derive the path latency depends on the number of tasks in the path and the size of the busy time interval of each task, i.e. the number k of possibly coinciding events in Equation 4.6 due to their input event model. In the worst case, all distributions of the coincidences on the local resources may be checked (e.g. in the example of Figure 4.1 with 2 relevant tasks and a maximum coincidence of 2 events, 3 busy time combinations are possible). In general, given a maximum number of b interfering events along a path of length l this leads to a maximum number of $\binom{(l-1)+b}{b}$ operations. In practice the number is smaller, because the slower tasks dominate over many candidates. In the given example that contains alternative paths, the total number of comparisons was between 31 in the case of no input jitter for $S0$ and 65 in the case where the jitter was 200 (which

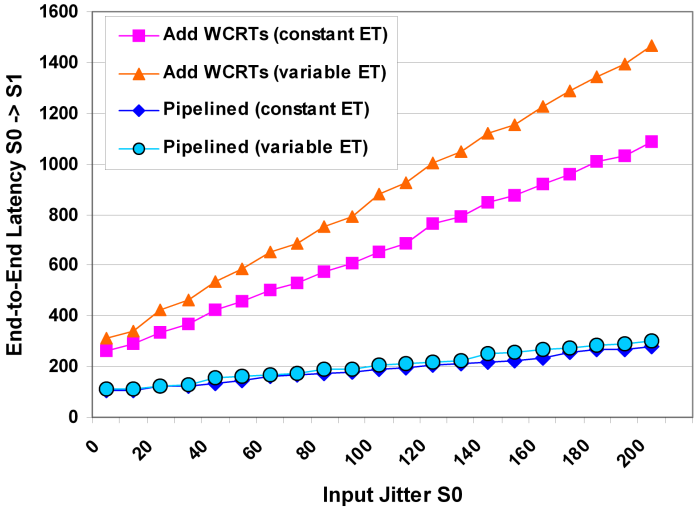

Figure 4.5: Comparison of Path Latencies.

corresponds to 10 coinciding events at the input).

4.6 Conclusion

This chapter provided a methodology to derive end-to-end path latencies in a multiprocessor system with heterogeneous components and complex application topologies. The method is suitable to consider arbitrary load event models and abstracts the scheduling behavior with the multiple event busy time model, which allows for heterogeneous combinations.

By tracking the latest possible times for events along the path, we can accurately capture the timing of pipelined processing, avoiding the overestimation of the simple response time aggregation. The method extends the scope from the worst-case of a single event to the combined processing of a sequence of events, and thus allows filling the gap between worst-case transient and long term behavior. For large sequences, the resulting latencies allow conservatively approaching the long term throughput. In the presence of fork and join application topologies, we accurately capture the timing of parallel paths by tracing the event timing back to the common root. In the presence of unrelated path-external inputs, we showed how to consider the activation delay by computing the maximum drift between arriving events. We covered functional cycles and non-functional cyclic dependencies. The latter is addressed by the iterative

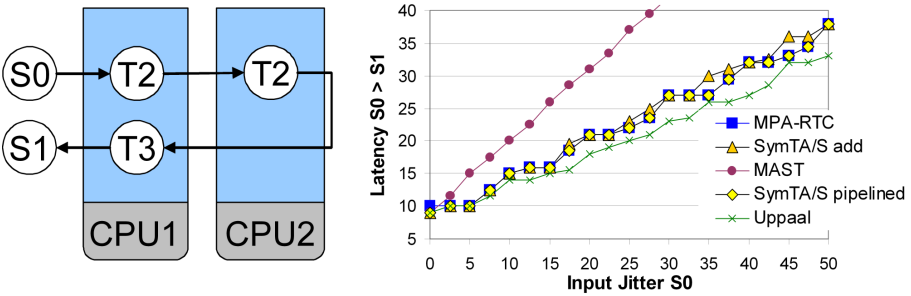

Figure 4.6: Example System and Comparison of End-to-End Latency (from [Per08]).

compositional analysis methodology that provides conservative task activating event models for each task. For functional cycles we provided a bound on the amount of past events that need to be considered because they may delay the processing of the investigated event. Besides the support for a large spectrum of application topologies, the analysis is also very efficient by relying on the multiple event busy time to provide a well bounded set of relevant timing candidates. Still, the experiments demonstrated that the derived end-to-end latencies are quantitatively on par with specialized approaches based on network calculus.

5 Shared Resource Request Bounds

We have seen in the introduction (Chapter 1) that the use of shared resources causes various new timing interdependencies. If tasks on two different processor cores compete for the same coprocessor, memory, or communication infrastructure, this makes an isolated analysis approach infeasible. We have already looked into orthogonalization approaches to avoid the interdependence in Section 1.5, but concluded that this approach is difficult to apply universally. Instead, the following methodology confronts the integration challenges by capturing the dynamic behavior with novel model parameters and providing corresponding analyses to allow for conservative predictions.

5.1 Recapitulation of the Analysis Procedure

The previous chapters have already established the key parameters for the analysis of systems with such dependencies: Firstly, we have shown how the global, holistic analysis can be decomposed into dedicated analyses for specific timing parameters in Chapter 2. Based on this, we provided improved predictions about the timing of task-activating events in Chapters 3 and 4. These analyses rely on the availability of the task's worst-case response time, or better, its multiple event busy time.

To provide this metric in the presence of dynamic resource timing and online arbitration, the analysis must recognize the correlation between the relevant analysis parameters. The analysis decomposition provided in Chapter 2 allows to entangle the set of dependencies shown in Figure 2.4. To extend the analysis of distributed multiprocessor systems for the treatment of shared resources, we particularly require the following three analysis components:

- The main concern is the response time, or better the multiple event busy time of tasks that can be delayed by incomplete shared resource operations. For this, we provide the "interdependent scheduling analysis" in Chapter 6.
- In order to deduce the task's timing, one has to bound the aggregate latency of a set of relevant shared resource operations. This is the focus of Chapter 7.
- The aggregate latency of the operations on dynamically arbitrated shared resources can only be investigated, if we know the load that is imposed on the shared resource from the competing tasks and processors. Analyses to derive this "shared resource request bound" are provided in the present chapter.

With these analysis functions, an intermediate analysis cycle can be embedded into the analysis procedure of [Ric03] shown in Figure 2.1. Whenever the response time

of a task is to be computed in this iteration, all three analysis functions have to be executed for the involved tasks and resources, in order to deliver the best estimate on the analysis results that can be derived from the current intermediate analysis state. The sequence of these sub-analyses is depicted in Figure 5.1.

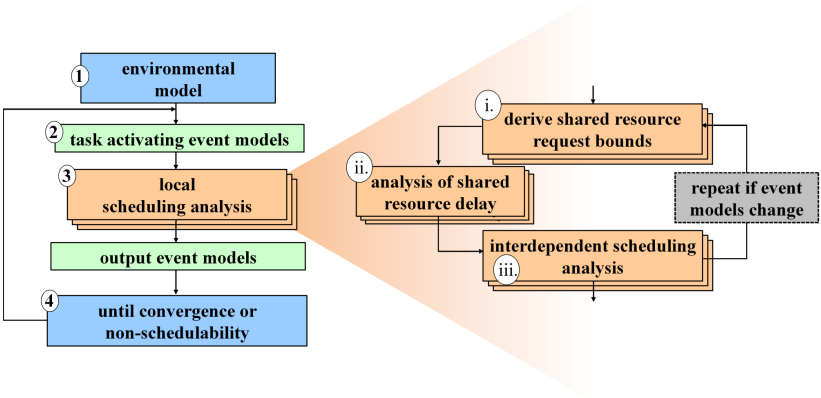

Figure 5.1: Shared Resource Analysis when Embedded into the Iterative Analysis Flow of Figure 2.1.

Of course, as Section 2.5.3 showed, the analysis results do not have to be updated according to this particular sequence. Any other analysis sequence will lead to the same fixed-point, as long as the analysis functions fulfill the conditions of Corollary 2.5. This will be the case for all analyses provided in the following chapters.

5.2 Related Work

To reduce the analysis complexity, previous work has suggested to either constrain the freedom of the run-time processor scheduling and resource arbitration, or to simplify the model of the request timing or the system state.

Considering Shared Resource Delays The resource arbitration can be set up in such a way that resources will be assigned to the different tasks independently of the current load imposed by other tasks in the system (see the discussion on countermeasures in Section 1.5). This reduces the online state space and allows an isolated verification for each task. For example, the memory bus can be scheduled according to a static time-driven scheme that provides an independent service to each processor. This however introduces inefficiencies, as many time slots that have to be reserved but will not be used. In [Ros07, And08] and [Sch09d] the worst-case cost of orthogonalization was reduced by deriving optimal bus schedules with respect to a worst-case memory

access pattern of each task. However, such approaches do not support local task scheduling and are thus applicable only as long as the number of tasks in the system does not outgrow the number of processors.

The shared resource service that is then not claimed by a task can be reused by other tasks. Such a behavior is provided e.g. by round-robin arbiters or other budget schedulers, which guarantee a lower bound on the service provided to the resource requests of each task (as investigated in [Hen07b, Sta09a]), and redistribute the left-over service according to a specific policy (e.g. based on priorities [Sta09a]). This combines the benefit of run-time isolation with run-time efficiency. However, the service guarantee that has been provided by previous approaches does not consider the actual amount of service re-use and assumes that all other tasks make full use of their budgets.

To investigate the timing impact of dynamically arbitrated shared resources, it was identified in [Sta05c, Sch06a, Sch06b] that load models are necessary to quantify not only the task activations but also the run-time load imposed on the shared resource. These load event models are the basis for performing an analysis of the shared resource delay as in [Sch06c, Sch08b, Sch10b]. In [Sch08b, Sch09b, Sch06a] it was shown that these delays can be included in the worst-case response time analysis also of dynamically scheduled tasks. An overview over the combination of these concepts can be found in [Sch09b, Sch10a].

The work in [Pel10a] focuses on the competition of peripheral data transfers to the shared memory with the execution of real-time tasks on a processor. The shared resource load of interfering operations was also characterized with event models, but the potential interference was only investigated for an execution trace. The approach was extended to multicore systems in [Pel10b] with a constrained preemption model (mainly to avoid cache thrashing) and time-driven/sequential superblock scheduling. By capturing the timing of interfering load with event models, conservative bounds can be provided for this setup.

The idea of constraining the task behavior and scheduling was further investigated in [Sch10c], where the dedicated phases are assigned within the superblocks for local execution and shared resource accesses and compares different scheduling options. The shared resource itself is arbitrated with a TDMA schedule. Instead of constraining the task behavior itself, one can also constrain the scope of the model: The analysis complexity is significantly reduced by performing the analysis only on execution traces (as in [Sta09a] and [Pel10a]) instead of capturing the complete behavioral state of the requesting task. However, the results are then not generally valid and may be overstepped at run-time.

The benefit of the work cited around [Sch10a] is that it provides safe worst-cases response times in the presence of local preemptive run-time scheduling and dynamic run-time resource arbitration. By relying on run-time load models, the framework can also be used to improve the response times in the presence of budgeted schedulers.

With this, also the run-time reuse of over-dimensioned budgets can be accounted (as shown in Section 7.3.2). This approach serves as the basis for the methods provided in the following chapters.

Note that the modeling of the shared resource performance with the service curve model provided by network calculus is not ideal to capture memory delays, because the service curve is only valid under the assumption that the requests are continuously backlogged [LeB01]. However, memory request for example are commonly initiated in disjoint time windows. The interpretation of a shared resource service curve would then lead to a minimum service assumption for each request (which we pointed out in Section 1.4.1 is not realistic). Instead, we introduce a new model in Chapter 5 that explicitly allows for such disjoint operations.

Estimating the Shared Resource Load If no orthogonalization measures as discussed above and in Section 1.5 are in place for the shared system resources, a model on the run-time load imposed on the shared resource from different components in the system has to be established. But only limited effort into this problem has been invested by previous research.

The timing of dynamic memory accesses has been the concern of [Sto05] which focuses on bounding the access times in PC-like architectures. The approach is based on simulation, and thus only of limited use in hard real-time systems. The derivation of conservative, application dependent resource request bounds for individual tasks was the concern of [Sch06b] and [Alb06], where the task's internal control flow was investigated. The basic assumption is that for each basic block the execution time is either constant or a minimum execution time and a maximum number of shared resource requests is known. Through program path analysis, distances between multiple requests are derived. We build on these approaches in this chapter and additionally derive bounds on the load imposed by sets of tasks.

Task-Level Synchronization Protocols On a higher level, shared resource operations can also be protected by operating system primitives that ensure mutual exclusion. Resorting to such mechanisms will avoid the physical competition for the shared resource, making the access times more predictable. The cost however is a higher task-level synchronization overhead.

Such synchronization protocols can be classified into *lock-based* and *lock-free*. In the former case, a task is only allowed to perform critical code once it has acquired a lock (a *semaphore*). In the latter case, it will begin the critical operation "optimistically" and only resort to a fallback mechanism when it has been disturbed. Both have different performance implications [Bra08b]. These algorithms can be efficient to operate on shared data objects. However, they can not be applied to protect the use of physical resources, where any disruption of an operation leads to undefined states. The importance of lock-based protocols is further underlined by the desire to maintain compatibility to single-core designs, where locks are intensely used [OSE05]

for safe sharing of data and resources. Also the multicore AUTOSAR OS provides rudimentary support for locks in its 4.0 release (latest version here: [AUT09]).

Lock-based protocols for multiprocessors go back to [Raj88], where a multiprocessor version of the priority ceiling protocol was proposed. Alternatives are for example the multiprocessor stack reuse policy (MSRP) [Gai03] and more recently, the flexible multiprocessor locking protocol (FMLP) [Blo07]. The corresponding analyses are mostly constrained to a simple sporadic task model, which can be inadequate as we illustrate in Section 5.3. In an empirical study, these and other algorithms have shown to have different strengths, with neither dominating the others on the complete spectrum of applications [Bra08a]. This finding is supported also by an analytical investigation of simple protocols in [Neg10]. We will discuss the consideration of these protocols in the response-time analysis of a task in Section 6.3.4.

Typical algorithms to assign the semaphores in the domain of single-processor systems ensure that a task instance may only *once* during its execution be blocked by lower priority tasks [Raj88]. In multiprocessor systems such a rigid bound is difficult to achieve because several tasks will execute in parallel, and are able to repeatedly lock a shared resource [Sch09b, Neg10]. Thus a task may find a resource locked each time it attempts an accesses. The blocking time in this setup is than a function of the number of resource requests issued by the analyzed task (and the equivalent number of its voluntary suspensions), as well as the amount of resource requests by any *other* component in the system. The shared resource load provided in Chapter 5 enables a reasoning about the blocking time in these setups. In Chapter 6, Section 6.1.1 we will revisit the related work to show how the resource delays can be considered in the busy time analysis.

5.3 Introduction

A limitation of previous approaches to capture the interference on a shared resource is the relatively simple model of the task's individual resource usage. For example, in [Raj91] a constant number of requests per task execution is assumed. The lack of research on more sophisticated models is owed to the fact that such bounds are not required to accurately capture the behavior of single processor systems, where the blocking time can be precisely calculated without knowing the tasks' exact request patterns. For example in a single processor system where shared objects are protected by the priority ceiling protocol, only a single critical section can block a higher priority task; this makes the exact request times insignificant. This however is generally not true for multiprocessor systems. Here, tasks on different processors that compete for the same resource will continue their execution after they have used the resource, which implies that they can later occupy it again. Thus each task may cause multiple conflicts as indicated in the example of Figure 1.5. It is therefore advisable to take a closer look at how the requests are timed.

This is illustrated in Figure 5.2 which depicts two tasks τ_1 and τ_2 on two different

processors competing for a share resource S. The resource arbitration is such that τ_2 receives a higher priority on the resource, thus conflicts are resolved in its favor. Now, assume τ_1 tries to access the resource 4 times during its execution. The same is true for τ_2. The exact timing of τ_2's accesses now clearly makes a difference. If all requests occur at the beginning of its execution (Figure 5.2a), this may cause τ_1 to be blocked on each access. If however, τ_2's requests are (and are known to be) further separated in time (Figure 5.2b), this means that during τ_1's execution, only one or two conflicts may actually occur.

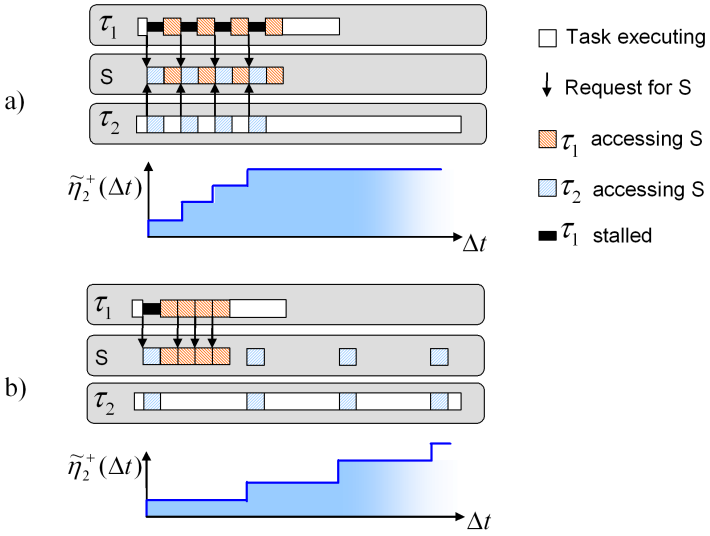

Figure 5.2: Relevance of Accurate Resource Request Bounds.

5.4 Modeling Refinement

In order to formally capture the behavior of tasks in a multiprocessor system with shared resources, we rely on an extended task model. This model is the basis for the considerations in the present and the following chapters.

5.4.1 Extended Task Model

The extended task shares the fundamental properties of the typical real-time task model (as in [Tin94b, Ric02b]). In the physical system, a task is *activated* by the expiration of a timer, the activation or finishing of a preceding task, or the arrival of a message. In our model, the respective conditions are modeled with *events* as

defined in Definition 3.1. Each *task activating event* creates a *task instance*. After a task instance was assigned to its processor for an amount of time that is equal to its worst-case execution time, the task is *completed*, and the task instance terminates. When more than one instance of a task has been activated (e.g. by a burst of incoming data), they are processed strictly in order, i.e. processing of an instance is commenced only after the preceding has terminated (out-of-order processing can be modeled with separate tasks). The resulting backlog becomes part of the tasks' response time.

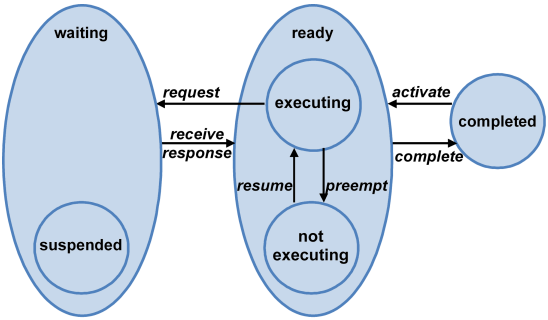

Figure 5.3: The Extended Task Model.

The different states of a task instance during execution are shown in Figure 5.3: An instance of an extended task can be either *ready* or *waiting*, which we formally define as follows:

Definition 5.1 (Task Instance Ready)**.** *A task instance is* ready*, when it has all data required to execute and can make progress with respect to its execution time when it is assigned the processor. A task is* ready*, when one of its instances is* ready*.*

Definition 5.2 (Task Instance Waiting)**.** *A task instance is* waiting*, when it has initiated a shared resource operation* (we also say: has issued a shared resource request)*, and can not continue execution before the operation has been completed.*

A *shared resource operation* consists of a fully ordered set of *requests* to a set of shared resources and is *complete* only after the requests were fully processed on each involved resource: i.e. transmitted via the bus, processed by the target, and the response was transmitted back to the requesting source. Definition 5.2 leaves open the possibility that the task continues to execute after issuing a request before it becomes waiting.

Each extended task instance is completed only when it has been assigned the processor for the execution time of that particular instance, and all of its requests have been completed.

In order to reason about the timing of a task's shared resource accesses, we are particularly interested in the following fundamental parameters:

- $\mu_{j \to S}^{max}$, the maximum number of operations the an instance of task τ_j requests from shared resource S
- $\tilde{d}_{j \to S}(n)$, the minimum requests distance between any n requests by an instance of that task to shared resource S.

In Section 5.5.1, we show how these metrics can be derived from a task specification. We will take a look at a special case in Section 8.1, in which these parameters can not be statically bounded at the task level, but are additionally influenced by the local scheduling and preemptions.

5.4.2 Capturing the Timing of Shared Resource Requests

To express the shared resource load from a given processor, we rely on the event models defined in Section 3.2, which provide an upper and lower bound on the number of events within a time window of given size. We denote the traffic to the shared resource with $\tilde{\eta}$ (and $\tilde{\delta}$) to differentiate it from task activating event models η (and δ).

Definition 5.3. *The* Shared Resource Request Bound $\tilde{\eta}_{T \to S}(\Delta t)$ *is the maximum amount of requests that may be issued from a set of tasks T to a shared resource S within a time window of size $\Delta t > 0$.*

The Shared Resource Request Distance $\tilde{\delta}_{T \to S}(n)$ *is the minimum time during which n requests may be issued.*

We already know from Section 3.2 that the shared resource request bound and the shared resource request distance are "pseudo-inverse" and can be converted to each other according to equations (3.7) and (3.5). We make diverse use of both representations based on convenience.

If T contains only one task (e.g. τ_j), we denote its shared resource request bound simply with $\tilde{\eta}_{j \to S}(\Delta t)$ and $\tilde{\delta}_{j \to S}(n)$; also the index S, denoting the shared resource, is sometimes omitted for brevity.

A requesting task may perform different types of operations to a shared resource. For example one type of operation may be a cache miss that results in the fetching of a memory row accessed as a burst of 8 words, and another type may be the fetching of a larger macro block in an image processing application. We model this behavior by allowing different implied shared resource processing times per requesting source. Further distinguishing the type of operations within a task execution leads to the problem of finding critical sub-sequences that incur the maximal load, which is exponential [Mok97]. We assume that larger operations are modeled as compositions of elementary operations (for example, the fetching of 1 macro block translates to 8 burst accesses).

5.5 Deriving Bounds on the Shared Resource Requests

This section first suggest several approaches to the derivation of the resource request bound of single tasks. Then the per-task results will be aggregated to provide the joint traffic issued by a set of tasks that are mapped to the same processor.

5.5.1 Remote Operations Initiated by a Single Task Instance

Let us assume for now, that all shared resource accesses are *explicit*, i.e. the result of special instructions in the source code or application binary. The timing of these requests is then dependant on the path through the task's internal control flow as determined by the value of the processed data. Implicit data fetches such as cache misses are more complicated to track, as they will surface dynamically during runtime and can not directly be identified in the application binary. This problem will be visited in Section 8.1.

5.5.1.1 Maximum Number of Requests per Task Instance

A relatively simple approach to reason about a task's shared resource request bound is by deriving the maximum number of requests per task instance — and neglecting the fact that there is a minimum distance between successive requests.

The number of requests per task instance can be bounded by investigating the task's internal control flow. For example, a task may fetch data each time it executes a for-loop that is repeated several times. By multiplying the maximum number of loop iterations with the amount of fetched data, a bound on the memory accesses can be derived. Focused on the worst-case execution time problem, previous research has provided various methods to find the longest execution path through such a program description with the help of integer linear programming (see [Li95, Li97] for the original proposition and [Wil08] for a historical overview over subsequent developments). By modifying the node weights to the number of request to a specific resource, this approach can be easily adapted to find the path with the maximum number of requests $\mu_{j \to S}$ per task instance (which may not necessarily be the path with the maximum execution time).

5.5.1.2 Minimum Distance between Successive Requests

Depending on the actual system configuration, relying solely on the upper bound on the number of requests per task instance may not be sufficiently accurate. In the analysis of the shared resource contention, this translates into an assumed burst of requests that may not occur in reality, because the task will actually execute for a certain amount of time between its requests. To more realistically capture the runtime load, it is worthwhile to identify a minimum time $\tilde{d}_j(n)$ that must pass in order for a task instance to produce n requests.

Let us first assume that there is a minimum distance between any 2 requests to the

shared resource which can be derived directly from the given setup. This can be due to a minimum number of instructions per issued request, or a communication function that only periodically polls the request buffer. A simple bound on the distance between any n requests can than be extrapolated according to the following lemma.

Lemma 5.1. *The minimum request distance $\tilde{d}_j(n)$ between any n requests issued by an instance of task j that performs a maximum of μ_j requests per instance is for $1 \leq n \leq \mu_j$ bounded by*

$$\tilde{d}_j(n) \quad = \quad (n-1) \cdot d_{sr} \tag{5.1}$$

where d_{sr} is the minimum distance between any 2 requests.

This approach works nicely for relatively simple processor pipelines (which are common in embedded systems). Here the minimum execution time for a sequence of instructions, such as a basic block without memory accesses, can be accurately lower bounded (e.g. by 1 cycle per instruction). However, common processor pipeline features such as superscalarity and out-of-order execution break this assumption, so that the minimum execution times of some instruction sequences approach almost zero.

Even in the presence of complex processor pipelines, the minimum request distance is also bounded independently of the application by inherent system properties. Assuming that a processor can have at most one incomplete operation on the shared resource, any 2 requests will be separated at least by the minimum time to service 1 remote operation in the system. This minimum time can usually be accurately lower bounded, because the involved operations on the bus and the memory have a well defined execution time even in the best-case scenario.

Improved application dependent request bounds can be provided by more closely investigating the task's internal control flow. The basic assumption is that for each basic block the execution time is either constant or a minimum execution time and a maximum number of shared resource requests is known (as in [Sch06b] and [Alb06]). Through program path analysis, guaranteed distances between multiple requests can be derived. This allows deriving meaningful distances, because longer chains of instructions can be considered (see [Sch06b] for a suitable algorithm).

As an alternative to a formal approach, the distance between successive operations can also be extracted through measurement. Industrial practice to reason about the timing of individual tasks in a system is extensive simulation in a virtual prototype model (as investigated in [Sch08d]), or measurement on an prototype implementation (with debugging tools such as [Mar08]). As the complexity of a single task is relatively small (compared to the complexity of a complete system), a sufficiently large set of stimuli can be derived that covers most of the possible state progressions of a task. Although this does not deliver definite guarantees, this approach has received considerable attention, in particular with respect to predicting the amount [Sch08b, Sta09a] or latency [Sto05, Cha03b] of dynamic shared memory accesses (see also Section 5.2)

5.5.2 Multiple Instances of the Same Task

An embedded system typically serves its purpose through the repeated invocation of its software tasks. The shared resource traffic of multiple instances of the same task will exhibit a hierarchical pattern that follows from its activation pattern and the request distances per task instance.

The following theorem bounds the minimum distance between requests from multiple instances of the same task. The idea is illustrated in Figure 5.4 and considers the following aspects. Firstly, we consider the distance between the required number of task instances (region "(ii)" in Figure 5.4). This distance is defined by the distance specified by the task's activating event model, minus the task's worst-case response time. The worst-case response time identifies the largest interval over which the requests of each instance may be distributed (also called the requests' jitter interval). Secondly, the theorem considers the fact that during the execution of each task instance a certain amount of time must pass in order for it to produce its requests as provided by Section 5.5.1.2 (regions "(i)" and "(iii)" in Figure 5.4). The resulting minimum request distance is denoted with $\tilde{\delta}^a_{j\to S}(n)$, because it considers the *a*ctivation distance of the involved task instances.

Theorem 5.2 (Minimum Request Distance Based on Task Activation Constraints)**.** *If a task τ_j is allowed to execute in such a way that each instance finishes earlier than R_j after its arrival, then the instances of this task will not produce more than n requests to a resource S in a time interval of size $\tilde{\delta}^a_{j\to S}(n)$:*

$$\begin{aligned}\tilde{\delta}^a_{j\to S}(n) \quad = \quad & \min_{1\le k\le \min\{n,\mu_j\}}\{ \\ & \tilde{d}_j(k) \;+\; \delta'_j(\lceil (n-k)/\mu_j\rceil + 1) \\ & +\; \tilde{d}_j(n-k-(\lceil (n-k)/\mu_j\rceil - 1)\cdot\mu_j)\;\}\end{aligned} \tag{5.2}$$

where

- *$\tilde{d}_j(k)$ is the minimum time that an instance of task j must execute in order to produce k requests to resource S (e.g. as provided by Lemma 5.1), μ_j is the maximum number of requests per task instance, and*
- *$\delta'_j(q)$ is the minimum distance between the execution of q instances of task j as given by its activating event model and worst-case response time:*

$$\delta'_j(q) = \max\{0, \delta^-_j(q) - R_j\} \tag{5.3}$$

Proof. Let all shared resource operations issued by task τ_j be numbered in the order of their request times. Assume two arbitrary requests m_1 and m_2 with $m_2 = m_1 + n - 1$. Let the task instances that produce request m_1 and m_2 be denoted with $i(m_1)$ and $i(m_2)$, respectively.

Let the instance $i(m_1)$ produce k of the n requests. It then must execute for at least $\tilde{d}_j(k)$ before it has done so. The remaining $n-k$ requests involve at least $\lceil (n-k)/\mu_j \rceil$ further task activations, because no instance can produce more than μ_j requests.

The distance between the activation of instances $i(m_1)$ and $i(m_2)$ is constrained by the task's activating event model $(\delta_j^-(n))$. Because $i(m_1)$ executes for no longer than R_j, $i(m_2)$ can not be activated sooner than $\delta_j^-(\lceil (n-k)/\mu_j \rceil + 1)) - R_j$ after $i(m_1)$ has finished, as provided in the second line of (5.2).

Assume that the intermediate instances produce the maximum number of requests μ_j each, then still $n - k - (\lceil (n-k)/\mu_j \rceil - 1) \cdot \mu_j)$ requests remain to be produced by instance $i(m_2)$. In order to produce the remaining amount of requests, the task must again execute for at least as long as demanded by $\tilde{d}$ as stated in the last line of (5.2). If the intermediate instances produced less than the maximum number of requests each, $i(m_2)$ would need to produce more remaining requests, which could only increase the distance between m_1 and m_2.

The instance $i(m_1)$ may produce any number k of requests that is smaller than both n and μ_j. The minimum over these scenarios is a lower bound on the actual distances. □

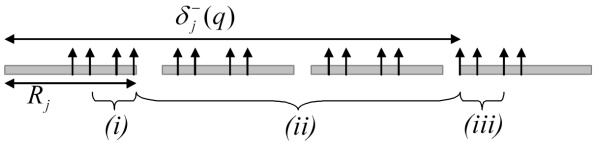

Figure 5.4: Illustrating Theorem 5.2 with $n = 14$ and $k = 3$. Region (i) shows the minimum distance between the $k = 3$ requests of task instance $i(m_1)$, region (ii) represents the minimum distance between the involved task instances, and region (iii) shows the remaining distance between the final requests.

In order to compute a task's requests distances according to Theorem 5.2 one needs to know the task's response time R_j. This of course may be unknown if this parameter depends on other aspects of the system (such as the interference by higher priority tasks).

To solve this, one could rely on other system properties to ensure a minimum distance between requests such as minimum round-trip times of the shared resource operations as defined by the hardware. Also, many previous approaches have assumed paradigms such as that all task activations are periodic, and their deadlines are smaller than the periods [Raj91], which allows for additional simplifications (and implies $R_j \leq D_j$).

However, in the present approach, we make no such assumptions. Instead, we have shown in Chapter 2 how the inter-dependant analysis parameters can be computed through iteration. When the response time of a task is refined, one has to recompute the respective minimum request distance. This procedure is possible, as long as all analysis components are co-monotonic, which we show in Section 5.6 for the approaches provided in this chapter.

Furthermore, the following Theorem 5.3 also bounds the request distance, but without relying on the task's response time. In this theorem, we identify the minimum amount of computation (i.e. processor occupation) that is required in order for a task to produce a certain number of requests. Then resulting minimum request distance is denoted with $\tilde{\delta}^{e}_{j\to S}(n)$, because it considers the minimum *e*xecution time between requests.

Theorem 5.3 (Minimum Execution Requirement Between Requests)**.** *In order to produce n requests to a shared resource S, instances of a task τ_j with best-case execution times t_{BC} must execute at least*

$$\begin{aligned}\tilde{\delta}^{e}_{j\to S}(n) = \ & \min_{1\le k\le \min\{n,\tilde{e}_j(t_{BC})\}}\Bigg\{ \quad \tilde{d}_j(k) \\ & + \quad \max\{0, \left\lceil \frac{n-k}{\tilde{e}_j(t_{BC})} \right\rceil - 1\} \cdot t_{BC} \\ & + \quad \tilde{d}_j\left(n-k-\left(\left\lceil \frac{n-k}{\tilde{e}_j(t_{BC})} \right\rceil - 1\right)\cdot \tilde{e}_j(t_{BC})\right)\Bigg\}\end{aligned} \tag{5.4}$$

where $\tilde{e}_{j\to S}(t_{BC})$ is the maximum number of requests per best-case execution time. It is derived from the minimum request distances $\tilde{d}_{j\to S}(n)$ as follows:

$$\tilde{e}_{j\to S}(t_{BC}) = \max_{n\ge 0}\{n \mid \tilde{d}_{j\to S}(n) \le t_{BC}\} \tag{5.5}$$

Proof. The proof follows along the lines of the proof for Theorem 5.2. The first, the last and the intermediate instances (lines 1, 3, and 2 of equation (5.4)) require a certain time to execute in order to provide the expected amount of requests. The highest density of requests can be observed if instances of the task execute "back-to-back", with each producing the maximum number of requests within its execution time. □

The previous two theorems deliver equally valid minimum distances between the requests of a task. It is worth noting that Theorem 5.2 makes no assumptions about the manner in which the task τ_j is actually *scheduled*, while Theorem 5.3 makes no assumptions about the manner in which the task is *activated*. Thus, if either of the required input parameters is lacking, the other theorem is still applicable. An optimal result is achieved by combining both results: Theorem 5.2 delivers more accurate results if task invocations are known to be separate in time, while Theorem 5.3 is more accurate if they may overlap.

5.5.3 Scheduling Multiple Tasks on the Same Processor

Tasks that share the same processor are executed alternately as directed by the local scheduling policy. This results in a combined request traffic $\tilde{\delta}^{-}_{T\to S}(n)$ for all tasks T mapped to the same processor. In this section we present two orthogonal lower bounds on the distances between the requests *per set of tasks* based on the analysis results *per task* that was provided in the previous section.

5.5.3.1 Minimum Distance Demanded By Task Execution Intervals

Theorem 5.2 has provided bounds on the distance between any n requests issued by instances of a task τ_j on the basis of the distance between the task activations and its worst-case response time, but independently of the amount of actual processing time that is assigned to the task. Consequently, this bound is valid for *any* actual schedule (as long as the task meats its worst-case response time, which is tautological).

The smallest time window in which n requests can be observed from a set of tasks T is trivially given by the smallest time window (Δt), in which the sum of the task's requests ($\sum_{j\in T} \tilde{\eta}^{a}_{j\to S}(\Delta t)$) is equal or larger than n. This is stated in the following theorem.

Theorem 5.4 (Request Distance per Task Set)**.** *The smallest time window in which n requests by tasks in a set T can be observed is larger than*

$$\tilde{\delta}^{a}_{T\to S}(n) = \min\{\Delta t \mid \sum_{j\in T} \tilde{\eta}^{a}_{j\to S}(\Delta t) \geq n\} \tag{5.6}$$

where

- *$\tilde{\eta}^{a}_{j\to S}(\Delta t)$ is the maximum number of requests of task τ_j to shared resource S in a time interval of size Δt (this value is derived from $\tilde{\delta}^{a}_{j\to S}(n)$ according to Theorem 5.2 and equation (3.5)).*

Proof. Follows immediately from the preceding reasoning. □

5.5.3.2 Minimum Distance Demanded By Exclusive Task Execution

Theorem 5.4 is conservative, but it can *under*estimate the actual request distances. This is owed to the fact that the tasks that are mapped to the same processor can not actually execute in parallel, but rather the scheduler will assign the processor exclusively to the different tasks over time.

The effect of this exclusive assignment is illustrated in the following example. Assume a set of tasks T is executing on the same processor and the tasks perform accesses to a shared resource S. In order for these tasks to produce a total of n requests, the tasks have to be scheduled in such a way that the sum of the requests by each task adds up to n (i.e. if n_j is the number of requests issued by task τ_j, we have

$\sum_{j \in T} n_j = n$). In order to produce n_j requests, task τ_j must execute for a certain amount of time, denoted as $e_j(n_j)$. Because at every point in time, only one task can be executed on a processor, the total time to produce the n requests is thus given by $\sum_{j \in T} e_j(n_j)$.

The following theorem formalizes this reasoning. We exploit the fact that each task must execute for a certain amount of time before producing its successive requests. According to Theorem 5.3 this execution time is lower-bounded per task by $\tilde{\delta}^e_{j \to S}(n_j)$, irrespective of the actual task state (i.e. the initial execution progress made with respect to a specific task instance).

Theorem 5.5. *If the tasks in a set T are scheduled alternatingly on the same processor, the smallest time window in which n requests to a resource S may be observed is bounded by*

$$\tilde{\delta}^e_{T \to S}(n) = \min \Big\{ \sum_{j \in T} \tilde{\delta}^e_{j \to S}(n_j) \mid \sum_{j \in T} n_{j \to S} = n \Big\} \tag{5.7}$$

where $\tilde{\delta}^e_{j \to S}(n_j)$ is the minimum time that instances of task τ_j must execute in order to produce n_j requests to resource S (as provided by Theorem 5.3).

Proof. For a total number of n requests to be issued by the tasks in the set T, the scheduler must select tasks for execution in such a way that the sum of the requests of the individual tasks is n. Thus, the problem is subject to the constraint $\sum_{j \in T} n_j = n$.

Because the processor is only assigned to one task at a time, the total time that needs to pass in order for the tasks in T to produce the n requests is given by the sum over the times that the individual tasks must execute in order to produce their respective share of the n requests ($\sum_{\tau_j \in T} \tilde{\delta}^e_{j \to S}(n_j)$).

Consequently, if the distribution of requests to tasks is such that this sum is minimized, the amount of time to produce the n requests is minimized. □

Finding the combination of requests per task that leads to a minimum overall request distance in equation (5.7) is an instance of the "bounded nonlinear Knapsack problem". Unfortunately, these problems in general belong to the class of NP-hard problems with exponential complexity with respect to the number of coefficients, i.e. tasks [Mar90].

The weight functions $\tilde{\delta}^e_{j \to S}(n_j)$ are usually not linear, but may at least be be *convex*, for example in the case where a task can issue a burst of requests in the worst-case, but then requests data with a regular rate. In this case, we can apply a greedy algorithm: Iteratively select that task for execution that requires the smallest incremental execution time to produce another request. But of course, the weight functions are in general not convex, but only *super-additive*. Here, we have several options: Either, we bound the actual request distance function by a convex one (which has been

done e.g. in [Lam09] and [Ste10] for task activating event models) — this is efficient but can lead to some overestimation. Or alternatively, we draw from the optimized algorithms available for the original problem (see [Li09]) to provide an exact solution.

5.5.3.3 Example

The individual shared resource request bounds of two tasks and the joint request bound derived on the basis of Theorem 5.5 are depicted in Figure 5.5. The most critical case is given if all tasks are in a state in which they are on the verge of producing another request, and the scheduler then briefly executes both tasks. It is then more critical to continue τ_1's execution until it has produced 4 requests. In order to produce another request however, τ_1 would have to execute relatively long, thus executing τ_2 becomes more critical (we have $\tilde{\delta}_1^e(5) - \tilde{\delta}_1^e(4) > \tilde{\delta}_2^e(2) - \tilde{\delta}_2^e(1)$). For comparison, the request distances provided by Theorem 5.4 are shown as a dotted line. Because the latter theorem does not consider the exclusive task execution it underestimates the minimum distance between requests in the given case.

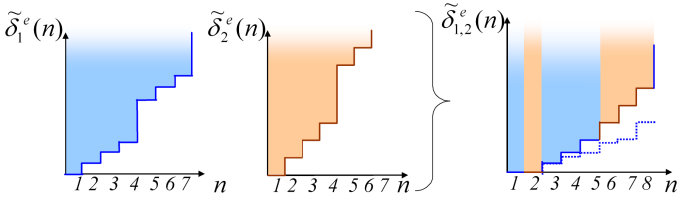

Figure 5.5: Joint Request Bound under Exclusive Execution.

The two theorems provided in this section both provide conservative distances between requests scheduled on the same processor. As with the "per task" theorems in the previous section, both make orthogonal assumptions: On the one hand, Theorem 5.4 is founded only the results of Theorem 5.2, which well captures the timing of multiple task instances that are separated over time. On the other hand, Theorem 5.5 relies on Theorem 5.3, which recognizes the fact that the tasks must execute for some amount of time in order to provide their requests. This execution time aspect is particularly important when the activations of multiple task instances can overlap.

The maximum of both analyses then provides the most accurate lower bound on the event distances:

$$\tilde{\delta}^-_{T\to S}(n) \geq \max\{\tilde{\delta}^e_{T\to S}(n); \tilde{\delta}^a_{T\to S}(n)\} \tag{5.8}$$

5.6 Embedding the Analysis Functions into the Multiprocessor Analysis

In order to embed the analysis functions provided in this chapter into the decomposed multiprocessor analysis procedure provided in Chapter 2, their behavior has to adhere to the conditions of Corollary 2.8. The condition demands that the analysis functions are monotonic with respect to their input parameters and that the domain of the analysis results forms a complete partial order.

Monotonicity

We first show that the computation of the minimum distance according to the theorems provided in this chapter are monotonic with respect to their input parameters.

Lemma 5.6. *The minimum request distance $\tilde{\delta}(n)$ computed according to Theorem 5.2 is monotonic with respect to its input parameters.*

Proof. The only relevant input parameters are the distance between task activations $\delta_j^-(q)$, the task's worst-case response time R_j, and the distance between requests per task instance $\tilde{d}_j(n)$. Let the modified parameter values be denoted with $\delta_j^{-\prime}$, R'_j and $\tilde{d}'_j$, and due to the monotonic progression of these parameters we have a possibly increased response time estimate $R'_j \geq R_j$, a possibly decreased distance between task activations $\forall n : \delta_j^{-\prime}(n) \leq \delta_j^-(n)$, and a possibly decreased distance between the task's requests $\forall n : \tilde{d}'_j(n) \leq \tilde{d}_j(n)$.

Because of $\forall n : \delta_j^{-\prime}(n) \leq \delta_j^-(n)$ and $R'_j \geq R_j$ we can quickly see that the distance between the task instances $\delta_j^{-\prime\prime}$ as computed with these parameters from equation (5.3) can not grow: $\forall n : \delta_j^{-\prime\prime}(n) \leq \delta_j^{-\prime}(n)$.

Obviously, if for any n: $\delta_j^{-\prime\prime}(n) \leq \delta_j^{-\prime}(n)$ or for any k: $\tilde{d}'_j(k) \leq \tilde{d}_j(k)$ then the summation in equation (5.2) can not become larger. Thus, also the minimum over all terms identified by k can not increase. □

Lemma 5.7. *The minimum request distance $\tilde{\delta}(n)$ computed according to Theorem 5.3 is monotonic with respect to its input parameters.*

Proof. Here, the relevant input parameters are the distance between requests per task instance $\tilde{d}_j$ and the task's best-case execution time t_{BC}. We assume the best-case execution time estimate to be constant, and not affected by the analysis progression: $t'_{BC} = t_{BC}$. Let the progression of intermediate analysis states lead to two sets of input parameters with $\forall n : \tilde{d}'_j(n) \leq \tilde{d}_j(n)$. Trivially, the first term in equation (5.5) can thus not become larger. Then, for equation (5.5) we have $\tilde{e}'_{j\to S}(t_{BC}) \geq \tilde{e}_{j\to S}(t_{BC})$, because no larger amount of requests may be observed during the best-case execution

time. Consequently, the second term in (5.4), the time spanned by the intermediate task instances, can not become larger.

The third term in (5.5) is more tricky. The parameter of the $\tilde{d}_j$-function may grow due to $\tilde{e}'_{j\to S}(t_{BC}) \geq \tilde{e}_{j\to S}(t_{BC})$. This adds additional candidates to the computation of the minimum function. However, per candidate x, we have $\tilde{d}'_j(x) \leq \tilde{d}_j(x)$, and thus, the third term may not increase for the initial candidates, and any additional candidates can not lead to an increase of the joint minimum. □

In Theorems 5.4 and 5.5, these per-task analysis results are aggregated to parallel or co-scheduled sets of tasks. The corresponding analyses are monotonic with respect to a decreasing distance of requests per task. This can be easily seen, because the analyses involve only order-preserving operations (minimum and addition).

Complete Partial Order

Next, we show that the domain $SRRB$ of the shared resource request bound forms a complete partial order.

Firstly, we can compare to analysis results with the comparator defined in Definition 3.5 for task activating event models. An analysis result $\tilde{\delta}'$ "is greater than" another $\tilde{\delta}$, if the minimum distance for any n, we have $\tilde{\delta}'(n) \leq \tilde{\delta}(n)$, i.e. the distance between requests is possibly smaller, which translates into an increased load.

Secondly, a smallest and a greatest element are defined by

$$\tilde{\delta}^{\top} : \tilde{\delta}^{\top}(n) = 0 \ \forall n \tag{5.9}$$

and

$$\tilde{\delta}^{\perp} : \tilde{\delta}^{\perp}(n) = \infty \ \forall n. \tag{5.10}$$

Based on these considerations, we can conclude that the analysis of the shared resource requests bounds as provided in this chapter complies with Condition 2.8 and can therefore be embedded into the compositional analysis approach of Chapter 2.

5.7 Summary

This chapter has recapitulated the procedure to tackle the inter-processor dependencies in a multiprocessor system with dynamically arbitrated shared resources. We have highlighted the need for accurate load models for the shared resource, and provided corresponding analyses that allow the derivation of the minimum shared resource request distance.

For this, we first investigated the task's internal control flow to derive the individual request distances during the execution. Then we turned to the behavior of multiple instances of the same task, taking into account their activation distances and intermediate execution requirements. In two theorems with orthogonal assumptions, we

introduced an analysis to extrapolate the aggregated shared resource load caused by multiple tasks. If the task set is mapped to the same processor, we considered the exclusive execution assumption to provide an accurate bound.

Finally, we established that the provided analyses comply with the conditions of Corollary 2.5, so that these analysis functions can be included in the iterative analysis procedure suggested in Chapter 2.

6 Interdependent Scheduling Analysis in the Presence Of Shared Resources

6.1 Introduction

The response time analysis of real-time tasks classically works on the assumption that once a task is activated, the corresponding instance has all data available in order to complete its execution, and will be kept from executing only by local scheduling interference. This model, which we assumed in Section 3.5, is however not sufficient to cover the case where tasks perform accesses to shared resources in multiprocessor systems.

In this chapter, we investigate the effect of such external operations on the tasks' response times. For this, we first present our analysis concept which differentiates between local and external contributors to the task's busy time. Then we look into two classes of shared resource operations — synchronous and asynchronous — which allow for different local scheduling decisions. For each setup we provide the analysis for a common scheduling policy that complies with these assumptions: that is, static priority preemptive scheduling in Section 6.2.1, and multithreaded round-robin scheduling in Section 6.3.2). We also note alternative procedures for synchronous and asynchronous operations (e.g. active waiting instead of stalling) and discuss these with respect to performance and predictability. In Section 6.3.4, we argue for the applicability of our analysis for the analysis of blocking times due to semaphore locks to protect critical sections. Finally, we show that these scheduling analyses fit into the decomposed multiprocessor analysis approach suggested in Section 2, and conclude in Section 6.5.

6.1.1 Analysis Concept and Related Work

As introduced in Section 3.5, many scheduling analyses are based on the busy window technique, in which the interfering workload on a processor is recomputed in an iteratively growing time window until the response time of a task has been found. This analysis procedure has been adapted to a wide array of academic and industrial scheduling policies: For example, static priority preemptive scheduling [Tin94b], non-preemptive scheduling with static and dynamic priorities [Geo96], budget schedulers [Beh07], OSEK [Kai07], FlexRay buses [Pop06], and more recently even global multiprocessor scheduling [Ber07, Gua09]) can be addressed. In addition to these numerous approaches to extending the scope of the busy window analysis, another

type of extensions is concerned with the accuracy of the analysis results, for example, by considering input-dependent task execution times [Mok97], generic task activating event models [Sch08c], or offsets between task activations [Tin94a, Hen06b].

In other work the analysis was extended to cover run-time effects beyond the simple academic assumptions of initial research in the real-time domain [Liu73, Jos86, Leh90, Tin94b]. Examples of such extensions include the analysis of the task synchronization delay due to blocking times of critical sections [Raj88, Raj91, Tin94b], realistic modeling of the context-switch overhead [Kat93, Jef93, Kai07], or considering the effects of competition for a local cache or other preemption-related delays [BM96, Lee96, Lee01, Pet01, Sta05b]. Also the impact of processor external operations (i.e. coprocessor calls) with constant access time has been quantified for single processor systems in [Kim95, Ble05].

The wealth of these approaches has shown that the busy window based analysis is able to conveniently accommodate extensions.

The run-time overheads investigated by these approaches invalidate basic assumptions about the task's core execution time and scheduling policy, and their presence inevitably leads to larger task response times. The cited solutions propose to reenable the busy window analysis by inclusion of additional *busy time contributor* terms in the workload equation. For example, the static priority preemptive analysis with some of the above overheads becomes:

$$w = q \cdot C_i + \sum_{j \in hp(i)} \eta_j^+(w) \cdot C_j + w_{block}(w) + w_{cs}(w) + w_{crpd}(w) \tag{6.1}$$

where $w_{crpd}(w)$ is the incremental busy time contribution by cache-related preemption delay in a time window of size w (from [Pet01]), $w_{cs}(w)$ is the maximum context-switch overhead (from [Kat93]), and $w_{block}(w)$ is the maximum blocking time due to PCP (according to [Raj91]). These terms, and the complete set of their relevant input parameters, are described in more detail in the literature cited above.

The key observation is that even though the different busy time contributors have been investigated in completely different contexts, their combined application is still possible. Of course, in order to ensure that the results are valid, a unified model has to be established and considered by all "subanalyses": For example, *dropping* the very common assumption of zero context-switch overhead implies that there is some operating system function involved. This function will in practice also load instructions from the memory, and either has to be considered in the analysis of the cache-related preemption delay, or to be locked in the cache. Similarly, the resource arbitration can introduce additional context switches when tasks are suspended, which have to be considered. But once a suitable model has been found, each subanalysis is able to conservatively deliver the respective incremental busy time contribution with a specialized algorithm.

This pattern of incrementally augmenting the response time analysis — or lately the multiple event busy time analysis — with new timing aspects can also be applied to the delays experienced due to external shared resource operations. For this, we capture the aggregation of the delays due to shared resource accesses is in the following *external busy time*:

Definition 6.1 (External Busy Time)**.** *The* external busy time *is the amount of time that the completion of instances of a task τ_i is delayed due to tasks on its processor (including itself) performing external operations.*

This definition is very general and because it does not refer to any actual scheduling properties does not give an indication on how the external busy time can be derived for a specific scheduler. We accordingly denote as the *local busy time* the total amount of time that a task instance is kept from completing due to the task itself executing, the local scheduling interference, and other previously studied delaying effects as cited above. Note that in light of this definition, the term *busy* window can be misleading, because the processor does not have to be physically occupied. With respect to the analysis of a task τ_i, we understand as the busy window $w_i(q)$ a time interval during which the q coinciding instances of τ_i are not complete. Following the above definition, the corresponding analysis can also be split into two logical components: An analysis of the local execution, local interference, and scheduling effects (the total time being denoted with w_{local}), and an analysis that provides the additional time during which an external shared resource operation hinders the progress of the investigated task ($w_{external}$). The total busy time is then given by

$$w_{total} = w_{local} + w_{external} \tag{6.2}$$

As in fact the local interference depends on the time window over which the execution is spread, and the interference on the shared resource depends on the time window over which the requests are distributed (see Chapter 7), the delay computation needs to be included in the iteration.

$$w_{total} = w_{local}(w_{total}) + w_{external}(w_{total}) \tag{6.3}$$

Equation (6.3) represents a fixed-point problem, because w_{total} appears on both sides of the equation. In order to be able to find a valid solution by iteration, the analysis components need to be monotonic with respect to their input parameters (see also Condition 2.8). If this is the case, the procedure suggested in [Jos86],[Leh90], and [Tin94b] can be applied: Beginning with an initially optimistic estimate on the total busy time, the busy time is iteratively recomputed according to (6.3) until a fixed-point has been found. This fixed-point then is a valid solution to the equation, and thus (if the equation is correct) a conservative estimate on the task's busy time. Alternatives to this procedure are discussed in Section 6.4.

The proposed analysis decomposition provides several major benefits that are similar to those known from compositional performance analysis of distributed systems: Firstly, it allows maximizing the reuse of existing scheduling policies and corresponding analysis techniques. Secondly, it decomposes the overall analysis problem into disjunct aspects that can be investigated separately. And thirdly, it reflects the composite structure of modern embedded system design, in which the scheduling and shared resource arbitration are separate concerns that are open to individual optimization and refinement.

6.1.1.1 Assumptions

For the scope of this chapter, we assume that all relevant input parameters are known. Thus, they either represent a priori knowledge as provided by the system analyst, or we embed our analysis into an iterative analysis procedure as defined in Chapter 2. Specifically, we assume:

- All data that would be required to perform the analysis of a task also if it was not sharing resources is available (i.e. task mapping, worst-case execution times, activating event models, the parameters of the scheduling policy,...)
- For each task τ_i, we have the shared resource request bound to each shared resource S, in particular $\mu_{i\to S}$, denoting the maximum number of requests to that resource per instance of τ_i. Also, if so required by the analysis, an analysis function exists that provides the maximum number of shared resource requests that may be issued by a set of tasks $p(i)$ mapped to τ_i's processor: $\tilde{\eta}^{+}_{p(i)\to S}$. The derivation of this value is the concern of Chapter 5.
- Finally, an analysis exists that allows to bound the total amount of time during which shared resource operations can be ongoing. This input parameter is captured by the *aggregate busy time*, the computation of which is the concern of Chapter 7.

Modern processor pipelines and memory architectures are becoming increasingly complex, posing several challenges such as timing anomalies to formal analyses [Kir08, Wil09]. To facilitate tight bounds, a certain degree of timing composability is required to constrain the state space that needs to be investigated to find tight worst-case execution time estimates. If not explicitly stated otherwise, we assume that each processor core has a timing-compositional architecture, in the sense that any shared resource delays are additive to the execution times. We discuss the implications of out-of-order execution in Section 6.2.2.

6.1.1.2 Treatment of Incomplete Shared Resource Operations

A task that seeks access to a shared resource, generally has two options on how to proceed:

a) The shared resource operation is performed *synchronously*. With this, we understand that the task can not continue executing before its resource request has been fulfilled. Furthermore, the scheduler will usually not take note of the ongoing operation, and receives no opportunity to reschedule in favor of another task during the resource operation.

b) Alternatively, the task can perform an *asynchronous* shared resource operations. In this case, the operation consists of a request that does not halt the task's progress, and a successive synchronization instruction (i.e. a *wait*-statement). Commonly this synchronization operation is interpreted by the scheduler in such a way, that the task can be *suspended* until its shared resource request can be fulfilled. This would allow other tasks to execute in the meantime.

In the following sections we investigate both setups. The optimal choice with respect to the tasks' timing depends mainly on the expected duration of an external operation, the available processor features (such as context switch overhead), and the amount of competition for the shared resources.

6.2 Synchronous Shared Resource Requests

From the point-of-view of processor and system design, the simplest method to offer shared resource operations to tasks is to provide a multi-cycle operation that stalls the complete processor until the operation has been processed. This procedure is the common practice for the case where the expected resource access time is short, the competition from other processors is low, and the system is applied in a cost-sensitive environment that in effect constrains the processor complexity.

The importance of this setup is underlined by the fact it is the most common mechanism to provide physical access to a system memory or memory mapped coprocessors (such as a floating-point unit). Some of the most commonly used embedded processor cores provide multi-cycle operations for this use-case, for example the ARM9 [Seg98] (as used in the Texas Instruments OMAP1 dual-core system-on-chip [Tex04]) and many PowerPC derivatives [Gar94]).

A possible resulting gantt diagram is shown in Figure 6.1. Two tasks, τ_{lp} with a low priority and τ_{hp} with a high priority, initially are *ready* to execute (according to Definition 5.1). τ_{hp} receives the processor due to its priority, and will request a shared resource S_a, and later a shared resource S_b. As the shared resources are also used by other tasks in the system, they are not immediately assigned to τ_{hp}, but only after they become available. As can be seen, because the processor stalls, the low priority task never receives the chance to execute, even when τ_{hp} can not make any progress. In this setup, it is very easy to identify the external busy time contribution $w_{external}$, which is the time spent waiting for a shared resource.

Stalling the processor has the advantage that the state of the processor's pipeline at the time of completion of the external access is highly predictable (it is the same

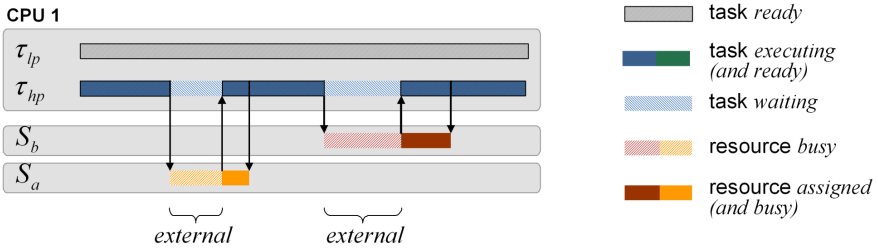

Figure 6.1: Tasks (τ_{hp} and τ_{lp}) sequentially access two shared resources (S_a and S_b); the processor stalls during the accesses.

as at the time when the operation was requested). As noted in [Kir08] and [Wil08], such "additive" behavior allows for an efficient verification procedure and increases the achievable accuracy. This will be exploited by the corresponding response time analysis in the next section.

6.2.1 Static Priority Preemptive Scheduler with Shared Resources

Let us now take a closer look at the timing behavior of the static priority preemptive scheduling policy, when applied on a processor platform that stalls during external operations. The definition of the scheduling behavior is as follows:

Definition 6.2. *The* Static Priority Preemptive Scheduling Policy *at any given point in time assigns the processor to a task according to the following rules:*

- *When no task is* waiting*, assign the processor to the highest priority task that is* ready*;*
- *When a task enters the* waiting *state, stall the processor until the external operation is finished, i.e. defer any scheduling decisions until the respective task is again* ready*.*

A task is said to be ready *or* waiting *according to Definitions 5.1 and 5.2.*

In order to reason about the task's external busy time, we can make the following observations: Firstly, no context switch can occur during an external resource operation, because the processor is then stalled. Secondly, whenever a request has been issued, no other external operation from the same processor can be pending, because it can not be issued when the processor is stalled.

The first observation implies that a *waiting* task is non-preemptable: If during an operation a *higher* priority task becomes *ready*, the higher priority task will be blocked. This can lead to a brief *priority inversion*. As there can be only one ongoing operation

(see second observation), the duration of this blocking is bounded by the maximum time to perform a single external operation. Once a higher priority task has been chosen for execution, no task with lower priority will have the chance to execute, and thus no further blocking may occur before the high priority task is completed.

These observations also apply to the case where the tasks on the investigated processor perform accesses to more than one shared resource. There will never be two incomplete parallel accesses to different shared resources from the same processor. This trivially ensures that there can be no deadlocks due to accesses to different shared resources (for nested locking of semaphores, see Section 6.3.4).

Still, as said before, investigating each shared resource operation individually would lead to unrealistic results. The following analysis is based on the observation that for the given scheduler, the busy time of multiple instances of a task is the sum over the times during which at least one of the instances is *ready* or *waiting*. The total ready time corresponds to the busy time of the scheduler without shared resources in Theorem 3.3, and the overall waiting time is bounded by the shared resource delays which we will investigate in Chapter 7.

Theorem 6.1. *Given a static priority preemptive scheduler that schedules tasks on a processor according to Definition 6.2, then q coinciding instances of a task τ_i are completed in any time window that is not smaller than $w_i(q)$ according to the following equation:*

$$w_i(q) = \underbrace{q \cdot C_i + \sum_{j \in hp(i)} \eta_j\big(w_i(q)\big) \cdot C_j}_{w_{local}} + \underbrace{\sum_{S \in \mathbb{S}_i} A_S\big(\tilde{\eta}_{p(i)\to S}(w_i(q)), w_i(q)\big) \quad + \max_{S \in \mathbb{S}_i}\{B_S\}}_{w_{external}} \tag{6.4}$$

where

- *C_i,C_j,$hp(i)$,$\eta_j(w_i(q))$ are as defined for Theorem 3.23;*

$p(i)$ *is the set of tasks consisting of task τ_i and all higher priority tasks $hp(i)$ that are mapped to the same processor ($p(i) = \{i \cup hp(i)\}$);*

$\mathbb{S}_i$ *is the set of all shared resources to which tasks that are mapped to τ_i's processor perform accesses;*

$\tilde{\eta}_{p(i)\to S}(\cdot)$ *is the shared resource request bound of the requests that are sent to the shared resource S from tasks in $p(i)$ (for example provided by the analysis according to Theorem 5.4);*

$A_S(\tilde{\eta}, w)$ *is the aggregate shared resource delay of these operations, i.e. the maximum amount of time during w at which one request in $\tilde{\eta}$ has been issued, but has not been completed (this corresponds to Definition 7.1, for which an analysis is provided in Chapter 7);*

B_S *is the blocking time inflicted by delayed context switches when the processor is stalled due to accesses to a shared resource S.*

Proof. Once q instances of task τ_i have been *activated*, they can be kept from being *completed* only due to three factors: *(1.) The processor is executing an instance of task i.* This can only be the case for a total time $q \cdot C_i$.

(2.) The processor is executing an instance of a higher priority task. The combined workload that higher priority tasks may accumulate in the busy window of size $w_i(q)$ is bounded by $\sum_{\forall j \in hp(i)} \eta_j(w_i(q)) \cdot C_j$.

(3.) The processor is stalled, because it is waiting for a shared resource operation to be completed. For case 3, two exclusive scenarios can be differentiated:

(3a.) The processor is waiting for an operation from a lower priority task to complete. This can only happen once, because as soon as an instance of τ_i has started executing on the processor, no lower priority task receives the opportunity to execute before all q coinciding instances are complete, and thus those tasks can not issue further requests. The longest non-preemptable time by a lower priority task is bounded by $\max_{S \in \mathbb{S}_i}[B_S]$.

(3b.) The processor is waiting for an operation from any other task to complete. This is the case whenever any task in $p(i)$ has issued a request to a shared resource but the operation is not complete. The number of requests to a specific resource $S \in \mathbb{S}_i$ performed by this set of tasks in the busy window of size $w_i(q)$ is bounded by $\tilde{\eta}_{p(i)\rightarrow S}(w_i(q))$. Their experienced aggregate busy time during $w_i(q)$ is given by $A_S(\tilde{\eta}_{p(i)\rightarrow S}(w_i(q)), w_i(q))$. The same reasoning applies to every shared resource $S \in \mathbb{S}_i$. Consequently, the total waiting time for operations by the tasks in $p(i)$ is bounded by the sum $\sum_{S \in \mathbb{S}_i} A_S(\cdot)$.

Each of these busy time contributors only holds true for the provided amount of time. After the union of these times has passed, all q instances of task τ_i must be completed. □

Equation (6.4) again represents a fixed-point problem in itself, with $w_i(q)$ appearing on both sides of the equation, that can be solved by iteration. This topic will be visited in Section 6.4, where we also argue that the right-hand side of (6.4) is monotonic with respect to all of its input parameters. By relying on this property the analysis can be embedded into the larger analysis framework provided in Chapter 2.

6.2.2 Discussion on Alternative Handling of Synchronous Shared Resource Operations

Stalling the complete processor during shared resource operations implies that no scheduling decisions can be made while the operation is ongoing. This leads to a blocking time for any local task that may become *ready* in the meantime (this results in the term $\max[B_S]$ in Equation (6.4)). When the length of the shared resource operation is relatively long or it may take a long time before the resource is assigned to the requesting task, this behavior can be unacceptable with respect to the response

time of higher priority tasks. We now briefly discuss design options that ameliorate this problem and give indications for their analyzability.

Active Waiting As an alternative to an explicit multi-cycle shared resource operation, the shared resource access can also be implemented through *active waiting*. In this scenario the task repeatedly polls the external resource until it is available. It will usually remain preemptable which reduces the impact on higher priority tasks. Moreover, the task does not suspend, which implies the overhead to resume its execution is low. It is relatively easy to implement even without extensive hardware support. However, this procedure can be most efficiently implemented in the presence of (global) atomic test-and-set operations, that are supplied in some multiprocessor implementations (e.g. the Freescale MPC56xx multicore derivatives for automotive powertrain applications [Fre09a], or the ST's StepNP platform [Pau02]).

The task's response time analysis is in this case straight-forward, because the shared resource delay ($w_{external}$ in (6.4)) is still additive to the task's execution time (as in the case where the processor stalls). Because the waiting delays the task's finishing time it can experience additional preemptions by higher priority tasks (as considered by w_{local} in (6.4)). The special case in which a preemption takes place during a waiting interval (and thus in parallel to the resource operation, causing a reduced impact on the response time) can usually not be guaranteed, and may thus not be assumed in the worst-case.

The most critical aspect of this procedure is the polling mechanism itself. It can only be safely applied in very controlled situations, because the unsuccessful access attempts cause an additional load on the communication infrastructure that may impact other time-critical applications. Deriving the resulting load on the shared resource and communication infrastructure is challenging (and as to the best of our knowledge not been formally addressed) in particular if no systematic scheme exists to ensure a bounded number of retries (for example through additional inter-task signaling).

Out-of-order execution To reduce the impact of the shared resource operations on the overall processor utilization, the processor can service workload that is not dependent on the completion of the external operation. Without context-switch, some processor pipelines perform *out-of-order execution* of instructions (e.g. the IBM PowerPC 601 [Bec93] or some ARM cores such as the Cortex A9 [ARM09]). This technique allows executing "future" instructions of the currently active task during the otherwise unused pipeline phases. While this procedure increases the achievable processor utilization, it has a less clear impact on the worst-case behavior.

Different aspects interact: On the one hand, some of the task's instructions will be processed in parallel to its external operation, which reduces the remaining execution time required by the task when it again becomes *ready*. On the other hand, if the distance between resource operations is determined by the progress of the task

execution (and not by system properties as discussed in Section 5.5.1.2), they can move closer together in time due to the potential task speed-up. This increases the load imposed on the shared resource, which will in effect lead to a decreased service available to *other* tasks in the system.

The gain of parallelism (first aspect) can to a certain extend be considered in the busy time analysis by subtracting the guaranteed parallel execution during each request from the execution time of the task, and in effect its response time (as suggested in [Kim95] and [Ble05]). The difficulty is that this parallelism is a run-time dynamic value differing for each access: The guaranteed amount of parallelism is only non-negligible if the corresponding resource access takes sufficiently long. Except for the case where the minimum resource access time is relatively large, the overall benefit is difficult to trace.

To avoid the implications of the second aspect, it is advisable to resort to system properties such as discussed in Section 5.5.1.2 to determine the distance between multiple requests. This could be the minimum round-trip time of a single operation, or a polling structure that periodically services shared resource operations. Relying on the best-case execution intervals alone would in the presence of out-of-order execution quickly lead to unsatisfactory values approaching zero in practice.

Similar challenges can also be observed for asynchronous shared resource requests, as will be noted in Section 6.3.1.1.

6.3 Asynchronous Shared Resource Requests

6.3.1 Implementation and Scheduling Options

To increase the achievable processor utilization the active task can be suspended while it is *waiting*, in order to let other (*ready*) tasks execute, even if they have a lower priority. To enable this behavior, the shared resource operation is split into a *waiting* phase and a phase where the resource is actually occupied. The requesting task can then be excluded from scheduling while its shared resource requests are pending. This procedure is particularly interesting when shared resource access times are relatively long (compared to the overhead to context-switch).

As an example, the OSEK operating system provides the *extended task* structure for this purpose [OSE05]. The possible states of such a task are depicted in Figure 6.2. In addition to the usual task states in preemptive scheduling, the task can also enter the "waiting" state when it waits for a specific event to occur. When in the waiting state, the processor can be assigned to other tasks, also of lower priority. The operating system is responsible to perform the necessary context-saving and restore operations. Note that the terminology used in this thesis deviates from the OSEK terminology but can be easily mapped: In the figure, the OSEK task states are superimposed on the states as defined in Definitions 5.1 and 5.2.

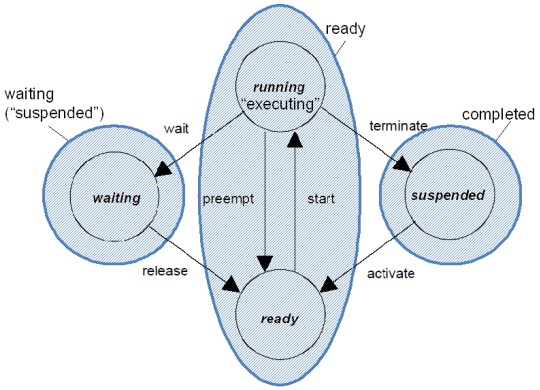

Figure 6.2: OSEK Extended Task Model (from [OSE05]) mapped to our Task Model.

The efficiency of the suspension procedure depends on the ability to perform fast context-switching. This is the motivation for some processor implementations to provide a set of hardware threads that allow performing a quick (sometimes single-cycle) context switch to another thread (e.g. [Mac98, Adi02, Pau02, Bek04]).

Figure 6.3 shows a possible schedule for the case where the shared resource is requested asynchronously. It differs from the example in Figure 6.1, such that the lower priority task τ_{lp} is allowed to execute during the times at which task τ_{hp} is *waiting* for the shared resource. Notice that no benefit can be observed concerning the response time of τ_{hp}.

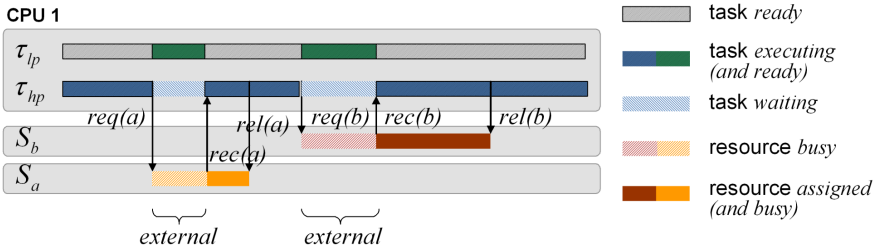

Figure 6.3: A high priority task accessing two shared resources asynchronously, giving the low priority task a chance to execute.

6.3.1.1 The Problem of Exploiting the Rescheduling Benefit For Real-Time Applications

In this setup the local processor load (i.e. average utilization) remains largely unaffected by the amount of shared resource usage, as long as there is another thread available that can be executed in the meantime. The impact of shared resource usage on the worst-case behavior however is more ambiguous and difficult to predict:

- On the one hand, we can observe that it is possible for some tasks (in particular with low priority) to finish earlier than in the case of a stalling processor. These tasks can re-use the processor cycles left over due to other tasks suspending.
- On the other hand, we can expect the same behavior that is also introduced by out-of-order execution (see Section 6.2.2): The requests by tasks on the same processor will possibly move closer together, which increases the load on the shared resource, and causes timing hazards for other involved tasks.
- Finally, if non-negligible, the additional context switching overhead must be taken into account.

Although the benefit of overlapping execution and shared resource operations that can be achieved with task suspension and multithreading may be observed quite commonly during a simulation run, the amount of parallelism that can be *guaranteed* to occur during any execution scenario is usually far less.

To illustrate this problem, Figure 6.4 shows a situation where a high priority task almost fully disturbs *both* the local execution and the remote operations of a lower priority task. The high priority task becomes ready for the first time just after the low priority task was activated. This imposes the local interference upon the response time of the low priority task. When the high priority task suspends due to an external operation, the low priority task can execute in parallel for a brief moment. But as soon as it issues its own request, there is no further parallelism, and its operation suffers the interference by the previously issued remote operation of the higher priority task. The local and the external busy time have almost no overlap. Given the specified task model, the depicted scenario may actually occur, and thus no parallelism can be guaranteed. Note that it would also be extremely difficult to uncover this scenario by simulation. Parallel execution is the main objective for using multithreaded execution, but as the example illustrates it is difficult to exploit its gain in a real-time system.

This effect was also observed in [Sch06a], where a priority based scheduler with multithreading and a first-come-first-served memory arbiter was investigated. The experimental results implied that there is no benefit for worst-case response times when multithreading is used in priority based multiprocessor systems. Worse, high priority tasks suffer from decreased response times, as their external operations will be subject to interference by lower priority tasks. Note that the experiments were concerned

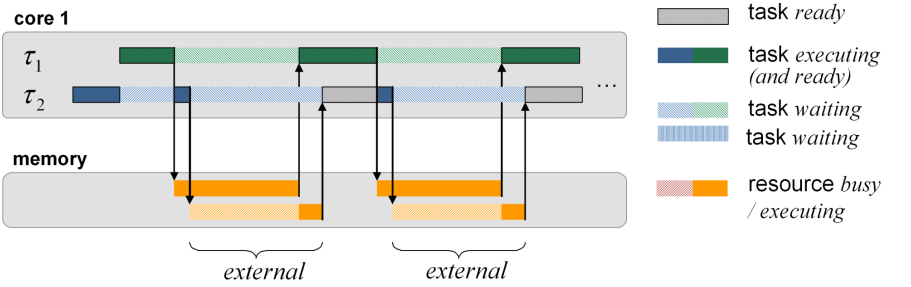

Figure 6.4: Example Execution Trace for a Task accessing the Shared Memory.

only with the worst-case response time; the observations can not be generalized to long-term behavior or throughput, for which multithreading is usually beneficial.

Other scheduling policies have also been suggested. In [Cro03] and later [Ive07] a coarse-grain cooperative round-robin scheduling was assumed and investigated, leading to the observation that the threads may even be starved, making it unusable for hard real-time systems. More predictable scheduling policies have also been proposed (see e.g. [Bar08a]), but the parallelization gain is generally only available for soft real-time tasks and the required hardware propositions have not found their way into actual systems.

For these reasons, the parallelism gain will not be reflected in the direct application of the decomposition analysis model presented in Section 6.1.1. However, if the specific setup allows for a guarantee on the attainable parallelism, this could be considered by means of a negative component.

6.3.2 Multithreaded Round-Robin

A common scheduling policy for deterministic real-time systems with symmetrical task significance, such as multiple parallel streams of data, is preemptive round-robin. Here, the processor is assigned to the different tasks in a static sequence while skipping those tasks that have nothing to process. A response-time analysis for such systems (without external operations) has been provided e.g. in [Rac07]. But this policy and analysis can not directly be applied when the tasks also accesses shared resources. In this section, we provide the busy time analysis for a particular case of thread scheduling in an embedded processor that was designed for network processing and multimedia applications [Pau02]. The corresponding setup will be revisited in Section 8.2 where quantitative experimental results are presented.

6.3.2.1 Basic Analysis of Multithreaded Round-Robin Scheduling

We approach the round-robin scheduling in two steps: First, we define the scheduling behavior and establish a worst-case performance analysis of the scheduler for a set of tasks without external operations. The basic lemmata will then be utilized in Section 6.3.3 to cover the case where the tasks perform accesses to a shared resource.

Definition 6.3 (Multithreaded Round-Robin Scheduling Policy)**.** *A uniprocessor which services a set of hardware threads* $\mathbb{T}$ *under the* Multithreaded Round-robin (MTRR) Scheduling policy *cyclically offers* t_{slot} *of execution time to each thread. Each thread that is not* ready *will be skipped with no additional overhead. As long as the number of tasks does not outgrow the number of hardware threads, all instances of a task* τ_i *are serviced by thread* i *in order of their arrival.*

Furthermore, we assume that the core execution time is a multiple of the slot size: $\forall i \in \mathbb{T} : C_i \bmod t_{slot} = 0$, which is the case in our motivational architecture in [Pau02]. The assumption can be dropped at the expense of over-approximations in Lemma 6.3 and Theorem 6.4 below,

First, assume that each task instance is continuously ready once it has been activated until it is completed (i.e. it does not initiate external operations and does not suspend itself). To derive the busy time function of a task instance, we will first bound the delay that its corresponding hardware thread can experience due to other threads executing. For this, we require a bound on the workload requested by other tasks, which we denote with $E_j(w)$, and provide afterwards.

Lemma 6.2. *Given hardware threads* i, j*, let thread* i *be ready for* s_i *arbitrary time slots within a time window of size* w*. Under MTRR, thread* j *may keep thread* i *from executing in this time window for no more than*

$$d_j(s_i, w) \leq t_{slot} \cdot \min\{s_i, \left\lceil \frac{E_j(w)}{t_{slot}} \right\rceil\} \tag{6.5}$$

where $E_j(w)$ *is the maximum amount of execution requested by thread* j *during* w*.*

Proof. For each thread j the amount of interference it may inflict on another thread (in our case, on thread i) is bounded by both a) the workload to be processed by thread j and b) the number of time slots offered to thread j. Both can be bounded by two simple observations:

a) Thread Workload: The maximal accumulated execution requirement for a thread j which services task τ_j depends on the time window size w and is bounded by $E_j(w)$ (see below for its calculation). Thus, thread j will have no remaining workload after it has received $\lceil E_j(w)/t_{slot} \rceil$ time slots, and will consequently waive any further time slots.

b) Thread Execution Opportunities: For each time slot for which thread i is ready under *MTRR* every other thread is offered no more than one time slot before i is assigned its time slot. Thus, s_i time slots of the same thread can be delayed by at most s_i time slots occupied per other thread.

Thus, the delay $d_j(s_i, w)$ that each thread j can cause to the s_i time slots of thread i within a time window of size w is bounded by (6.5). □

An important observation from this lemma is that the time slots of the investigated thread i can be distributed over the time window *arbitrarily*, i.e. task τ_i does not necessarily have to be ready for a *consecutive* sequence of time slots in order for Lemma 6.2 to hold. This is an important property for the analysis of tasks with external operations that will be exploited in Section 6.3.3.

Lemma 6.2 requires a bound on the amount of workload imposed by each interfering task τ_j. When the task τ_j does not suspend during its execution, its accumulated workload in the time window w, $E_j(w)$, can be bounded by the tasks worst case execution time C_j and the maximum number of activations defined by its activating event model $\eta_j^+(w)$:

$$E_j(w) = \eta_j^+(w) \cdot C_j \tag{6.6}$$

If however, as we will assume in the next section, the task τ_j may indeed suspend, i.e. enter the waiting state for a certain amount of time, this introduces the possibility that the execution of its workload is distributed over a larger time interval. Consequently, task instances of τ_j that have been before the beginning of the investigated time may have been waiting and then still have pending workload, while their new workload arrives as in (6.6). The magnitude of the shifted workload is bounded by the task's response time. With this, the maximum amount of execution requested by thread τ_j during w becomes

$$E_j(w) = \eta_j^+(w + R_j) \cdot C_j \tag{6.7}$$

Equation (6.7) will be referred to again in Section 6.3.3. Next, we we bound the number of time slots that instances of task τ_i occupy in order to be completed.

Lemma 6.3. *Assume a task τ_i with a worst-case execution time C_i that is mapped to thread i. Under MTRR, q coinciding instances of τ_i are completed after thread i has been assigned s_i consecutive time slots of size t_{slot}:*

$$s_i = q \cdot \lceil C_i / t_{slot} \rceil \tag{6.8}$$

Proof. Each instance of task τ_i can occupy the processor no longer than C_i before it is completed. This will be the case, when the instance has received $\lceil C_i/t_{slot} \rceil$ time slots. Because the q task instances coincide (i.e. the each instance arrives before the preceding is finished) and is never waiting, the task will occupy consecutive time slots. The total amount is given by the equation (6.8). □

From the preceding reasoning, we can now deduce τ_i's busy time function.

Theorem 6.4. *The maximum busy interval $w_i(q)$ of q non-suspending instances of a task τ_i scheduled under MTRR is bounded by*

$$w_i(q) = t_{slot} \cdot s_i + \sum_{j \in \{\mathbb{T} \setminus i\}} d_j(s_i, w_i(q)) \tag{6.9}$$

where

- $\mathbb{T}$ *are the processor's hardware threads; all instances of each task τ_j are serviced by thread j; t_{slot} is the duration of each time slot;*
- s_i *is the number of time slots required by the task instances from in thread i (bounded according to Lemma 6.3);*
- $d_j(\cdot)$ *is the delay imposed by a hardware thread j during the busy interval $B_i^+(q)$ (bounded according to Lemma 6.2).*

Proof. The q instances of τ_i require s_i time slots (according to Lemma 6.3), which will take no more than $t_{slot} \cdot s_i$ to finish. The processor arbitrating the hardware threads $\mathbb{T}$ allows each *other* thread $j \in \{\mathbb{T} \setminus i\}$ to delay the execution of thread i by at most $d_j(s_i, w)$ according to Lemma 6.2, and thus at most $d_j(s_i, w_i(q))$ within τ_i's busy interval. Thus, the multiple event busy time of q instances of task τ_i fulfills (6.9).

□

The right hand side of (6.9) is monotonic with respect to its input parameters, because it is a composition of order-preserving operations. Again, we can argue as in Lemma 3.15 that also its least fixed-point, $B_i(q)$ is monotonic. This enables its derivation either directly (as in [Jos86]), or in the context of the concurrent refinement of other system parameters as suggested in Chapter 2.

6.3.3 Multithreaded Round-Robin with Shared Resources

We now consider the additional busy time contribution imposed by the use of shared system resources. In the StepNP setup investigated in Section 8.2, the shared resource is a memory that can be accessed via a crossbar, such that the memory itself is the only point of arbitration. The external operations will cause voluntary suspension of the tasks, which will result in the corresponding thread to be skipped until the requested data is available.

Assume that the task requires data from the shared memory during its execution. Let each instance of τ_i perform no more than μ_i requests. The requests will separate the task instance into $\mu_i + 1$ subsequences of time slots during which the task is continuously *ready* with arbitrary lengths $s_{i,0}, s_{i,2}, ...s_{i,\mu_i}$. The sum of these subsequence

lengths constitutes the original number of occupied time slots:

$$s_i = q \cdot \lceil C_i / t_{slot} \rceil = \sum_{n=0}^{q \cdot \mu_i} s_{i,n} \tag{6.10}$$

Of course, not every task instance always performs the maximum number of requests, for example due to data-dependant behavior. In this case μ_i and s_i can be defined via the maximum number of request of q consecutive task instances. We assume that shared resource request are performed at the end of a time slot, which corresponds to the setup in [Pau02], where the time slots are assigned for single instructions only. With this, we define the following theorem.

Theorem 6.5. *The maximum busy interval of q instances of a task τ_i scheduled under MTRR that are* ready *for a total of s_i time slots and are intermittently* waiting *while performing no more than μ_i accesses per instance to a shared resource is bounded by*

$$w_i(q) \quad \leq \quad A_S(\tilde{\eta}_{pa(i)\to S}, w_i(q)) + t_{slot} \cdot s_i + \sum_{j \in \{\mathbb{T} \setminus i\}} d_j(s_i, w_i(q)) \tag{6.11}$$

where $A_S(\tilde{\eta}_{pa(i)\to S}, w_i(q))$ is the aggregate latency of τ_i's requests (based on the request bound $\tilde{\eta}_{pa(i)\to S}$ of all tasks $pa(i)$ mapped to the same processor as τ_i).

Proof. Any instances of τ_i will be in the waiting state for a time denoted with $W_{i,n}$ while its n-th shared resource operation is being processed (i.e. its data is requested over the bus, provided by the memory and transferred back to the CPU). After the operation is complete, the requesting thread has to wait for the next time slot in order to be serviced. It is then continuously ready for $s_{i,n+1}$ time slots before it requests the next data and becomes waiting again. Let the time to be serviced each of these time slots be $R(\cdot)$. Then, the q instances of τ_i are finished after:

$$w_i(q) \quad = \quad \sum_{n=0}^{q \cdot \mu_i} (R(s_{i,n}) + W_{i,n}) + R(s_{i,q \cdot \mu_i + 1}) \tag{6.12}$$

$$= \quad \underbrace{\sum_{n=0}^{q \cdot \mu_i + 1} R(s_{i,n})}_{\tau_i \ ready} + \underbrace{\sum_{n=0}^{q \cdot i} W_{i,n}}_{\tau_i \ waiting} \tag{6.13}$$

a. Bounding the time that τ_i is ready. We know that by definition all *ready*-subsequences of task τ_i will be serviced within $w_i(q)$. With this, the total ready time of τ_i can be bounded by Lemma 6.2 (which makes no assumptions about consecutive occupation of time slots):

$$\sum_{n=0}^{q \cdot \mu_i + 1} R(s_{i,n}) \leq t_{slot} \cdot s_{i,n} + \sum_{j \in \{T_j \setminus i\}} d_j(s_i, w_i(q)) \tag{6.14}$$

Recall that Lemma 6.2 requires the interfering workload of the considered thread, denoted with $E_j(w)$. If the task τ_j does not suspend itself, its this is bounded by equation (6.6), if it does suspend itself due to external operations it is bounded by (6.7).

b. Bounding the time that τ_i is waiting. It now remains to compute the intermediate waiting times $\sum w_{i,n}$ in Equation (6.13). We are looking for the "total amount of time within the busy interval w, during which at least one request from τ_i has been issued but is not complete". This corresponds to the definition of the *aggregate busy time* in Definition 7.1 which is derived in Chapter 7. Let the result of this analysis be $A_S(\tilde{\eta}_{pa(i)\to S}, w_i(q))$.

$$\sum_{n=0}^{\mu_i} w_{i,n} \leq A_S(\tilde{\eta}_{pa(i)\to S}, w_i(q)) \tag{6.15}$$

This allows us to expand equation (6.13) by calculating the local execution and ready times according to (6.14) and bounding the sum of the waiting times with (6.15), leading us to the theorem. □

The right hand side of equation (6.11) is again monotonic with respect to its input parameters, most critically $w_i(q)$, but also $d_j(\cdot)$ and $A_S(\cdot)$, because it is a concatenation of order-preserving operations (addition and multiplication). The same is true for the input parameters itself: $d_j(\cdot)$ as specified in equation (6.5) is monotonic with respect to the E_j, which again grows with increasing event load η_j^+ (decreasing activation distance) and task response times R_j. The monotonicity of the aggregate busy time computation $A_S(\cdot)$ is shown in Section 7.5.

Thus, the theorem qualifies for the evaluation according to the fixed-point theory either directly when all parameters are known, or in the context of the multiprocessor analysis as suggested in Chapter 2.

6.3.4 Task Synchronization Through Semaphores

Conflicting accesses on physical resources such as coprocessors or memories are typically resolved by the physical implementation of the bus arbiter, which signals the occupation or availability of the resource and assigns it to the processors in a run-time deterministic fashion. In addition to this "physical" arbitration — or in the absence of such — the shared resources can also be protected with logical semaphore locks. The exclusive access is than ensured through a synchronized behavior of the local scheduling policies.

Although this setup is not the primary concern of our decomposed analysis approach, we will see in this section that it can be adapted to cover this scenario. The argumentation for this ability is based on the observation that the *blocking time* due to task synchronization protocols is also a *response time contributor* that is usually

investigated in isolation and included additively in the response time analysis (compare [Sha90, Raj88, Sch09b, Blo07, Bra08a] and Section 6.1.1).

The basic synchronization concept is that a task which needs to access a shared resource first acquires a "lock", which is granted only to a limited number of tasks (e.g. one) at a time. When the resource is not available at the time of requesting the lock, the requesting task can be suspended until it is assigned the resource. This corresponds to the *waiting* state of the task model in Figure 6.2. We have briefly discussed suitable lock-based synchronization protocols in Section 5.2.

Considering Delays of Task-Level Shared Resource Synchronization The proposition of our analysis is to decouple the concerns of local scheduling and the arbitration of shared resources. If the synchronization protocol introduces modifications of the scheduling parameters at run-time (like the boosting of a task priority) this separation is challenged. We now take a closer look at the different task states and the resulting external busy time in order to show that the concerns of local scheduling (as investigated in this chapter) and remote arbitration (as investigated in Chapter 7) can still be separated in the presence of non-nested and also nested semaphore accesses.

If each task is constrained to locking only one resource at a time, this results in scheduling diagrams as depicted in Figure 6.3. When the task requests a resource, it becomes *waiting* and can be suspended by the local scheduler. The external busy time in this example is still bounded by total time that a task is *waiting*, i.e. it has issued a request to the shared resource, that has not been granted. Note that the time of the shared resource operation itself (the *critical section*) does not contribute to the *external* busy time, because at that time, the task is *ready*.

Nesting of Critical Sections Tasks may also be allowed to acquire more than one lock at the same time, which is called *nesting* of critical sections. This immediately introduces the concern of deadlocks, for example due to inverted access order from different processors. This can be avoided either by imposing a global order in which the tasks can acquire the locks (which then has to be communicated and to every software developer) or by the application of a synchronizations protocol that ensures deadlock-freeness through the temporary assignment of ceiling priorities (such as M-PCP). This intervention into the scheduling parameters causes an entanglement of the *local* and *external* busy time. This is illustrated in the following example.

Consider the asynchronous shared resource accesses depicted in Figure 6.5. Assume that the shared resource S_a is of crucial importance in the system, and any task that accesses this resource will receive a temporarily heightened priority above all other *ready* tasks on its processor during its access (this procedure is for example inherent in PCP and M-PCP). It may then happen that a lower priority task requests accesses to this resource, but is preempted by the investigated task τ_i before this access was granted. Then, as soon as the access is granted, the priority of τ_{l_p} is boosted, and τ_i may not execute although it is *ready*. Even though this "preemption" occurs locally,

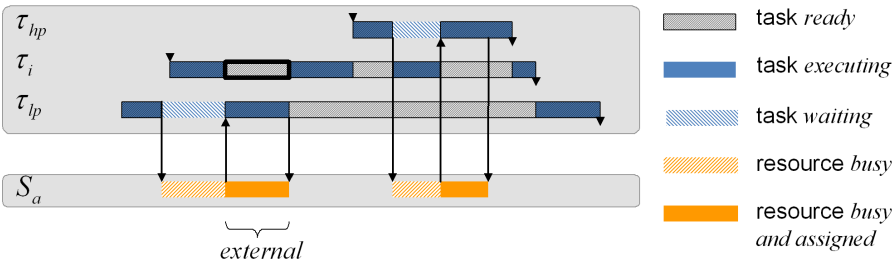

Figure 6.5: Tasks accessing a Shared Resource. Task τ_i experiences intervals of local and external busy time.

it contributes to the external busy time of τ_i, because it is an effect of re-prioritization due to shared resource usage. In addition, shown at a later point in the diagram, there is the classical interference by higher priority task preemptions, which would not have occurred if τ_i had not been blocked by the lower priority task.

Thus, in this setup with "static" priorities, the external busy time is the total time in which task τ_i is *waiting*, *plus* the maximum amount of time that it is *ready* but can not execute due to re-prioritization of a lower priority task (the critical section by the higher priority tasks is included in the local busy time). A detailed analysis of this bound on the external busy time for e.g. M-PCP has been provided in [Neg09] and [Sch09b]. Further discussion about the implications of different design choices for definition of the synchronization protocol, such as task preemptability, nesting, re-prioritization, and other aspects can also be found in [Neg10].

6.4 Embedding the Task's Busy Time Analysis into the Multiprocessor Analysis Procedure

From the scope of the system level analysis proposed in Chapter 2, computing the fixed-point of equation (6.4), and (6.11) is only one possible procedure to derive the values of the system's timing parameters.

As we have highlighted in before (for example in Section 2.4.1, the parameters needed to compute the task's response times are initially not known if they are the result of run-time execution variance and scheduling decisions. In particular, this is the case for the timing of the preemption interference due to e.g. higher priority tasks (η_j^+ in (6.4)), and the various possible conflicts from other processors on the shared resources that influence the value of the aggregate request latency $A_i(\cdot)$. Based on the argumentation in Section 2.5.2 and the respective chapters, these parameters will also evolve monotonically increasing during the analysis procedure. To allow the process

to converge, the computation of the busy time function in equations (6.4) and (6.11) has to adhere to the requirements provided in Condition 2.8.

As demanded by the first condition, the busy time analysis of the component τ_i is monotonic with respect to all influencing analysis parameters. We deduce this property from the observation that for both equations the right hand side only features order-preserving addition and multiplication operations.

The second condition demands that the intermediate analysis states form a complete partial order. As the busy time is scalar, this property is provided by defining the bottom element $B_i^{\perp}(q) = 0$ as a valid initial analysis state, and $B_i^{\top}(q) = \infty$ as the top element. Of course, the top element will never be reached by the analysis, as the analysis will stop as soon as the response time estimate lies above the deadline.

6.5 Summary

In this chapter we have looked into extending the task's busy time analysis in the presence of busy time contribution due to shared resources. We suggested to make the composite nature of many previous analyses explicit and introduced an *external busy* time to capture the timing impact of external operations.

We investigated two classes of shared resource operations: *synchronous* and *asynchronous*. For both scenarios, we investigated a representative scheduling policy: Static priority preemptive scheduling that stalls during shared resource requests, and a multithreaded round-robin scheduler that can handle asynchronous external operations. This also showed that the analysis decomposition is feasible for priority as well as non-priority based scheduling. By example, we demonstrated that the decomposition does not yield to overestimation with respect to the worst case, even when potential overlap is not accounted. Additional quantitative evaluation can be found in Chapter 8.

In addition, typical setups that deviate from the defined behavior were surveyed with respect to their analyzability. We also looked into mechanisms for task-level protection of shared resource accesses, and identified the busy time contributors that maintain the proposed analysis partitioning.

Finally, we showed that the analysis procedure defined in this chapter complies with the necessary conditions to integrate into the multiprocessor analysis approach introduced in Chapter 2.

7 Latency of Shared Resource Operations

7.1 Introduction

In the previous chapter, the task busy time analysis was revisited and extended to the case in which tasks perform operations on shared resources. The present chapter provides reliable latency bounds for these shared resource operations that contribute to the tasks response time. Our analysis concept as recapitulated in Section 5.1 provides a convenient decomposition of the concerns: The analyses provided in this chapter for different shared resource arbitration policies can be combined with any processor scheduling policy, as long as the latter supports the extended task model (which of course is the case for the ones considered in Chapter 6).

In the presence of dynamic online resource arbitration, such as for example the common first-come-first-served policy, the actual duration of the requests by a certain task heavily depends on the interfering load imposed from each other source of requests in the system. Our analysis will take this interference into account. For this, we rely on the shared resource request bounds per task, per set of tasks, and per processor as provided in Chapter 5.

The chapter is organized as follows: In the following section we identify the latency metric that is required for the extension of the response time analyses according to Chapter 6. We then provide two analyses: Firstly, we provide a simple bound based on the established worst-case assumption (Section 7.2) and then turn to the proposed enlarged analysis scope: For this, we derive a general analysis for work-conserving resource arbitration (such as first-come-first-served) and a dedicated analysis of round-robin arbitration in Section 7.3).

Next, the set of investigated system setups is extended to cover the case where each shared resource operation requires processing on a set of intermediate resources (such as a bus and the memory) in Section 7.4 and provide a toolbox of methods to tailor the analysis to a given setup. Our goal for the analysis is to integrate the results into the system level analysis of Chapter 2, which is established in Section 7.5. The analysis is than evaluated in Section 7.6. We conclude with a short summary in Section 7.7. We have already discussed related work in Section 5.2.

7.1.1 Capturing the Aggregate Latency Of Shared Resource Operations

The extensions of the task response time analysis proposed in Chapter 6 rely on the *external busy time* as a new contributor to the task's busy time.

As noted in Section 6.3.4, the external busy time consists of up to two components:

a) *direct blocking*: The total amount of time, during which the task can not make progress due to an ongoing external operation.
b) *indirect blocking*: The additional overhead incurred by changes to the local schedule.

The indirect blocking component highly depends on the utilized task scheduling policy, bounding its impact is therefore the concern of the inter-dependent scheduling analysis (as presented in Chapter 6). The total direct blocking time is however solely a property of the system components that are involved in processing the requests. Thus this metric allows isolating the system-wide timing impact of a task's shared resource operations. It can be bounded by the total time during which a request has been issued but has not been completed. We formally define this metric as follows:

Definition 7.1. *The* Aggregate Busy Time *of a set of operations to a shared resource is given by the aggregate amount of time during which at least one operation has been initiated but is not completed.*

As we will see later, this definition also drops the predominant focus on the worst case timing behavior of single requests, and provides the opportunity to investigate combined scenarios that span multiple events.

Although similar in name, it should be noted that the definition of the *aggregate busy time* is notably different from the *multiple event busy time* as defined in Definition 3.6. The latter applies only the the processing time of consecutive events that arrive and are processed in a continuous busy window, while the former captures the total processing time of a set of events that may or may not occur in disjoint time intervals.

7.1.2 Intractability of Exact Solutions

A key problem for the analysis of the aggregate latency of a set of operations is the very large space of possible system states at the time of the requests. This easily introduces major complexity problems, which has classically hindered the successful integration of such system operations into local timing analyses (see also [Wil09] for a discussion of this problem). As an example, recall the setup of Figure 1.5, where the accesses of the two tasks on CPU0 are disturbed whenever they coincide with the (periodic) accesses from CPU1. Small variations in the request times can lead to a significantly different execution time. For example, if the first preemption was only slightly shorter, not a single request from CPU0 would experience a conflict.

The exact time point at which a specific request occurs depends on a tremendous amount of online influences (the time of the task's activation, its input data, the processor's scheduling history, ...). This is also true for the timing of any competing accesses from other processors in the system, leading to a virtually unbounded state space. Despite the difficulty of harvesting the required information and formalizing

the possible behavior, it is an intractable challenge to consider such details in a conservative analysis.

While the application of a detailed timing model that covers all possible run-time behavior leads to a virtually unbounded set of candidates and thus entails an unacceptable analysis time, it also does not necessarily lead us to tighter bounds on the aggregate delays. In the end, a scenario may be identified as the worst case, in which most of the potential conflicts actually do occur. The larger the state space, and the more dynamic the system setup (for example in the form of request jitter), the less likely it is that one can rule out such a coincidence of events, which thwarts the benefit of the detailed consideration.

7.2 Analysis of the Aggregate Request Latency

7.2.1 Sum of Worst Cases

The predominant focus of real-time system's research is the derivation of the worst-case behavior of single events (for example, the worst-case task response time based on [Leh90] for single-processor systems, or the latency bound in network calculus [LeB01]). The existing analyses can be easily exploited to derive the aggregate delay of multiple shared resource operations, simply by assuming the worst-case scenario for every request. This approach has been followed previously for example in [Mün00] for the analysis of shared resource operations (such as database transactions).

In Section 3.5, we have introduced the *multiple event busy time* as a performance metric for events that are processed on a single processor. As opposed to the closely related worst-case response time, the busy time is defined independently of the actual timing correlation of the processed events (recall that we can easily derive the worst-case response time from the multiple event busy time with equation (3.22), and estimate the multiple event busy time from the worst-case response time with (3.26)). We will now rely on this metric, as it holds more detailed information about the resources' response behavior to a set of coinciding events.

7.2.1.1 Latency of Disjoint Shared Resource Operations

Based on the multiple event busy time model, on can easily extrapolate the aggregate busy time for a set of disjoint operations on a shared resource. Let $B^{+}_{p\to S}(q)$ be the busy time function of the shared resource S that processes the requests sent from a processor p. Furthermore, assume for now that the processor issues only 1 request at a time (for example because it stalls as soon as a request is sent). It can then easily be seen that every request must be completed after the single-event busy time:

$$A_{p\to S}(1) \leq B^{+}_{p\to S}(1) \tag{7.1}$$

This reasoning is valid for every request, so that we can straight-forwardly deduce

for q requests, that their aggregate busy time is bounded by

$$A_{p\to S}(q) \leq q \cdot B^{+}_{p\to S}(1). \tag{7.2}$$

Note that assuming $A_{p\to S}(q) \leq B^{+}_{p\to S}(q)$ would *not* conservatively capture the aggregate latency of the q requests, because the individual requests of coming from the processor p do not necessarily fall into a continuous busy window as demanded by Definition 3.6.

7.2.1.2 Latency of Potentially Overlapping Shared Resource Operations

To achieve a more efficient utilization of the requesting processor and also the shared resources, some processor implementations support the issuing of more than one shared resource operation simultaneously. This can for example be achieved with processor features such as out-of-order execution or multithreading. This setup was also investigated in Section 6.3.2, where we were concerned about the task's response times. This behavior breaks the reasoning of equation (7.1) because a request that is issued before the preceding operation has been completed is not necessarily finished after $B(1)$. Instead it will have to wait for the backlogged requests to be processed first — which leads to an increased latency.

Assuming q^{max_open} is the maximum number of open requests that may be issued from processor p to a shared resource S, and let q be again the number of requests for which we want to bound the aggregate request latency. Then we know that the first of the n requests will be finished no later than $B^{+}_{p\to S}(q^{max\_open})$ after it has been issued.

The remaining $q-1$ requests may each either be issued while their direct predecessor is still being processed, or afterwards, in which case it "opens" a new disjoint busy interval. Because we have $\forall n : B(n) \leq n \cdot B(1)$ (see Section 3.5.4.2), no scenario exists in which the aggregate busy time is larger than in the case where all requests are issued in separate busy intervals. Thus, we have:

$$A_{p\to S}(q) \leq B^{+}_{p\to S}(q^{max\_open}) + (q-1) \cdot B^{+}_{p\to s}(1) \tag{7.3}$$

7.2.1.3 Discussion

The shortcoming of the analysis approach based on individual requests is that it assumes overly pessimistic system states at the moments when the requests are issued. Not every request will be subjected to a worst-case system state, such as worst-case time wheel positions in TDMA, or transient overloads in priority based arbiters. Thus, not every operation will experience the maximum busy time $B(1)$.

For illustration, turn again to the example in Figure 1.5, where the accesses of the two tasks on CPU0 are disturbed whenever they coincide with the accesses from CPU1. A conflict will in this setup lead to the "worst-case memory access time",

but of all accesses from CPU0, this only happens 3 times in the example. Thus, accounting this interference for every memory access can lead to a prohibitive amount of overestimation.

7.3 Extended Scope Interference Analysis

By broadening the scope to a larger set of events and time interval, the *aggregate busy time* provides the opportunity to more realistically consider the effect of incidents that are relatively rare.

7.3.1 Bounding the Aggregate Busy Time For Work-Conserving Arbiters

The following theorem bounds the aggregate busy time for any work-conserving resource arbiter. Such an arbiter will not idle as long as there is work, i.e. a request, to be processed. This particularly covers priority-based arbitration, first-come-first-served, and round-robin scheduling, but not static time-driven schedulers (TDMA), where time slices may pass unused. The set of requests sent to resource S from a processor r in a time window of size w is denoted with $\tilde{\eta}_{r\to S}(w)$, which is the aggregation of the requests by all tasks mapped to processor r as provided in the Section 5.5.

Theorem 7.1 (Aggregate Busy Time of Work-Conserving Arbiter)**.** *Assume a work-conserving arbiter on a shared resource S. Then the aggregate busy time of q operations to S that are issued from the set of tasks on a processor r within a time window of size w is bound as follows:*

$$A_{r\to S}(q,w) \quad \leq \quad q \cdot m_r + \sum_{p\in P(r)} \tilde{\eta}_{p\to S}(w) \cdot m_p \tag{7.4}$$

where

- *q is the number of requests sent from processor r. m_r is the amount of time the shared resource S requires to process one of these requests.*
- *$P(r)$ is the set of* other *processors in the system from which tasks perform requests to the shared resource S.*
- *$\tilde{\eta}_{p\to S}(w)$ is the maximum number of requests sent by tasks on processor p within a time window of size w, and m_p is the amount of time required to process each of these requests.*

Proof. At any given point in time, the shared resource either (a) processes a request from processor r, or (b) from another processor, or (c) it is idling. As the arbitration is work conserving, the shared resource is idling only when there are no outstanding request, and thus the idle time (c) does by definition not contribute to the aggregate busy time. Thus the aggregate busy time is given by the total amount of time spent in either state (a) or (b). The total time in state (a) is bounded by the work implied

in the requests from r: When the arbiter has chosen to process requests from r for a time $q \cdot m_r$, all requests must be completed.

The total time in state (b) is bounded by the work implied by the interfering requests. The amount of requests that may arrive during the time window of size w, or have arrived earlier and are not completed is bounded by $\tilde{\eta}_{p \to S}(w)$. Each request implies work of no more than m_p. Thus, all requests from other processors are completed when they have been processed for $\sum_{p \in P(r))} \tilde{\eta}_{p \to S}(w) \cdot m_p$. Hence the total time spent in state (a) or (b) is bounded by equation (7.4). □

As Theorem 7.1 bounds the aggregate time during which at least one shared resource operation is incomplete, it delivers the upper bound on the aggregate request latency. It can be integrated into the tasks' response time analysis as explained in Chapter 6.

The theorem makes the simple assumption that all requests from a specific processor require the same amount of computation time on the shared resource. This assumption can be easily dropped by differentiating different kinds of requests and their implied amount of work. Care must than however be taken to find a conservative sub-sequence that leads to the maximum interference. This problem has been identified already for sequences of task activations, and modeling and analysis solutions have been suggested in [Mok97, Jer04, Wan05]. The arbiter could also distinguish between different streams, for example through prioritization of the requests of one processor over those of another, or by reserving budgets for different sets of tasks (as in [Sta09a]). This behavior can be easily captured by the analysis of the aggregate busy time mainly by modifying the set of processors $P(r)$ in equation (7.4) that may impose interference.

7.3.2 A Dedicated Analysis for Round-Robin Arbitration

As an example for a dedicated analysis for a specific policy, we present the analysis for round-robin arbitration of the shared resource. Under this policy, which is very similar to the scheduling policy for processors in Section 6.3.2, the shared resource is cyclically assigned to the different request sources, while those sources that have no pending requests are skipped. While in Section 6.3.2 we investigate for this policy the *multiple event busy time* or *worst-case response time* of the (coinciding) task invocations, in the present section we provide the *aggregate busy time* of the operations that are potentially separate in time.

Theorem 7.2 (Aggregate Busy Time of Round-Robin Arbiter)**.** *The aggregate busy time of q requests sent from a source r to a shared resource S that is arbitrated with a round-robin policy is bounded by*

$$A_{r \to S}(q, w) \quad \leq \quad s_r \cdot t_r + \sum_{p \in P(r)} \min \left\{ s_r; \left\lceil \frac{\tilde{\eta}_{p \to S}(w) \cdot m_p}{t_p} \right\rceil \right\} \cdot t_p \qquad (7.5)$$

with

$$s_r = \left\lceil \frac{q \cdot m_r}{t_r} \right\rceil \tag{7.6}$$

where m_r, m_p, $P(r)$, and $\tilde{\eta}_{p \to S}(w)$ are defined as in Theorem 7.1 and t_r and t_p are the time slot size assigned to requests from sources r and p. We assume that the work units are a multiple of the slot size: $m_r \bmod t_r = m_p \bmod t_p = 0$.

Proof. The proof follows the argumentation of Lemma 6.2 which applies to arbitrarily (as opposed to consecutively) requested time slots: Each interfering request stream can keep the requests from r from being serviced no longer than a) their induced workload, and b) their received opportunities to execute. □

Note that the assumption that work units are a multiple of the slot size is not unrealistic, in particular when all work units have the same size. If this is not the case, equation (7.6) has to be modified in order to be conservative, which may entail overestimation.

Using round-robin arbitration provides a safe and efficient system design. The bounded number of time slots offered to each source ensures that there is a limited interference between the processing of the different operations. This leads to a convenient verification procedure, in which the different request streams and their associated tasks can be investigated in isolation. But usually the task will not always completely exhaust its assigned time slot (or budget), which leaves more service available to other tasks. As noted in Section 5.2, the dynamic run-time reassignment is often not considered in the analysis, which leads to overestimated results. Theorem 7.2 however takes this effect into account by acknowledging the maximum amount of requested workload per task in equation (7.5).

7.4 Resource Requests over multiple Hops

Quite commonly, a resource that is shared by two or more processors is not directly accessible by these, but rather made available only via a bus or other interconnect framework with its own intermediate arbitration. In the previous section, we have assumed that there is only a single point of arbitration that needs to be acquired in order to access the resource. This is a matching assumption also in the case of a shared memory that is connected via a data bus, as long as the bus is occupied during the complete memory operation.

However, the shared resource operations can also be organized in the form of "split transactions", in which case the bus is not occupied during the complete operation (for example in [Pau02, Pau04, Ben02]). Rather, it will only be used during a "request phase" and is then available for other operations until the "response phase". It is also not uncommon that the request and the response are sent via physically different

routes and can not interfere with each other. These design options result in the following observations:

- A shared resource operation can involve sequential processing on multiple resources (i.e. an address bus, a shared memory, and a data bus).
- A different set of interfering operations may be experienced on each involved resource (i.e. the bus may be shared with all other processors, but the memory may be exclusively assigned and always be available).

To generically address this setup, the following sections provide a "toolbox" that allows tailoring the shared resource delay analysis to the specific system architecture.

7.4.1 Analysis Toolbox

First, we extend our model to the case where the shared resource operations issued by a task imply communication or computation on multiple independently arbitrated components. The set of involved components is captured in the following path:

Definition 7.2 (Shared Resource Request Path)**.** *Each shared resource operation from a specific source (processor) p to a shared resource S implies sequential in-order processing along a* request path *of resources $\mathbb{P}_{p\to S}$ that excludes the requesting processor itself, but includes the target of the operation.*

7.4.1.1 Aggregate Busy Time over multiple Resource Hops

A shared resource operation that requires processing on multiple resources will only be complete once it has sequentially been scheduled on each node along its request path. Thus, from the perspective of the requesting task, the request has an overall busy time that is given by the sum of the individual busy times on each resource.

Section 7.2 has provided a means to compute the aggregate busy time $A_{r\to S}(q, w)$ for a set of requests from a source r to a *single* shared resource S. This value is conservative for each resource along a shared resource request path, because the analysis makes no further assumption about the requests' timing relations. We can therefore bound the aggregate busy time of a set of operations on multiple resources as follows:

Corollary 7.3. *The aggregate busy time of a set of q requests sent within a time window of size w that require processing on resources V along a request path $\mathbb{P}_{p\to S}$ is bounded by*

$$A_{r\to\mathbb{P}_{p\to S}}(q, w) \leq \sum_{V\in\mathbb{P}_{p\to S}} A_{r\to V}(q, w) \tag{7.7}$$

The corollary does not make assumptions about the actual timing relation between the individual requests, and the assumed scenario ultimately corresponds to the case in which all request are issued in disjoint time windows. If however, the distance

between requests can be such that a request may be issued before the preceding is finished, this will actually result in a certain amount of pipelined processing. Given a known (maximum) distance between requests, this pipelining effect could be covered with the methods provided in Chapter 4, which would give us the maximum latency of a set of n requests along the request path.

7.4.1.2 Distortion of the Shared Resource Request Stream

When a set of shared resource operations is processed along a request path, the temporal relation between the requests will become distorted due to local scheduling and variable processing delays. This effect is well known from the behavior of tasks and messages in a distributed system and suitable analysis approaches have been covered in Chapter 3. In particular Section 3.6 has provided the derivation of event distances on the basis of a task's *activating event model* and the task's local *busy time function*. The same procedure can be used to track the possible distortion of the shared resource request streams. At the beginning of a path, the distance between the requests is given by the reasoning provided in Chapter 5. From there, the request event models at the inputs of the successive path elements can be consecutively derived with the known event model propagation procedure:

Corollary 7.4. *The minimum distance between requests of a shared resource request stream at the input of the $n+1$-th resource of the shared resource request path $\mathbb{P}_{p \to S}$ can be derived from the minimum distance between the requests at the input of the n-th resource together with its busy time function according to Theorems 3.11, and 3.12.*

The maximum distance between requests is provided by Theorem 3.13.

Recall that the event distance functions $\tilde{\delta}$ and the event load functions $\tilde{\eta}$ can be converted to each other according to equations (3.7) to (3.6). Thus the above corollary can also be used to bound the load imposed by interfering request streams.

7.4.1.3 Superposition of Request Streams that Coincide on the same Resource

When the shared resource operations involve computation on multiple resources, it can experience interference from a different set of request streams on each hop. This will for example be the case when the memory bus is shared by all processors, but not every request goes to the same memory or coprocessor. The competition for the shared resource node will introduce mutual delays into the respective latencies.

Theorem 7.1 relies on bounds on the amount of interfering requests in order to derive the aggregate busy time of the set of investigated requests. To apply this theorem in the case of resource request paths, the different event load functions of each interfering request stream need to be accumulated. This is provided by the following corollary:

Corollary 7.5. *Let $Paths(S)$ be a set of shared resource request paths in the system that involve requests on a shared resource S, and let the number of requests that each*

interfering event stream $ies \in P(S)$ *imposes on the shared resource* S *be bounded by* $\tilde{\eta}_{ies,S}(w)$ *per time window* w*. Then the combined load of the request streams in* $Paths(S)$ *is bounded by*

$$\tilde{\eta}_{Paths(S)}(w) = \sum_{ies \in Paths(S)} \tilde{\eta}_{ies,S}(w) \tag{7.8}$$

In addition to this rather simple approach there is a wide spectrum of approaches that can be exploited to better model the behavior of correlated request event streams. Chapter 5 has already investigated the effect of exclusive execution. Other approaches such as [Pal98, Hen06b, Kol06], and [Rox10] address the implications of potential offsets between the occurrence of successive events . It should be kept in mind that if such offsets can be observed (and shall be considered) this will usually lead to a set of load candidates, each of which have to be analysed to find the most critical scenario.

7.5 Monotonocity

While the aggregate busy time model already represents a useful performance metric in itself (for example to determine the experienced shared resource service that is provided to a certain processor), the ultimate goal is to apply it within the performance analysis of a complete multiprocessor system.

The analysis proposed in this chapter can be integrated into the tasks' multiple event busy time analysis according to e.g. Theorem 6.1, and into the iterative system level analysis of Chapter 2. The iterative analysis approach will converge towards a conservative fixed point, if the conditions that have been identified in Corollary 2.8 are fulfilled for each involved analysis function. This is the case for the proposed aggregate busy time analysis due to the following reasoning:

- The aggregate busy time is a scalar, and as such comparable, with a minimum and maximum value (0, and ∞). Thus, the analysis results form a complete partial order.
- The aggregate busy time is a function of the number of requests to the shared resource, the size of the time window within which these requests are issued, and the amount of interference by competing request streams within the specified time interval. It is monotonic with respect to these parameters, which can straight-forwardly be seen from e.g. equation (7.4), which is only a concatenation of order-preserving functions (addition and multiplication).

Consequently, the analysis of the aggregate busy time can be integrated into the system level analysis approach presented in Chapter 2.

7.6 Experimental Illustration and Evaluation

To evaluate the benefit of the aggregate busy time model over the analysis of individual worst-case scenarios, consider the following example. Assume a system with two tasks on two processors that access a shared resource. The investigated task τ_1 performs between $q = 1$ and $q = 10$ disjoint accesses somewhere within a time window of size $w = 500$, while the interfering task τ_2 accesses the shared resource in bursts of 5 accesses every 200 time units ($\tilde{\eta}_2^+(w) = \lceil w/200 \rceil * 5$). Each request requires a total of 20 time units to be completed.

From this follows that the worst-case access time for a single access by τ_1 is given by solution of $w = 1 * 20 + \tilde{\eta}_2^+(w) * 20$, which is $w = 120$. According to equation (7.2), the aggregate busy time is then conservatively approximated with $A(q) \leq q * 120$. Resorting to the larger time scope to analyze the aggregate busy time discussed in Section 7.3 allows capturing the bounded possible interference by τ_2. The aggregate busy time according to Theorem 7.1 provides the following bound: $A(q, w) \leq q * 20 + \tilde{\eta}_2^+(w) * 20$, which for a time window of size 500 is $A(q, w) \leq q * 20 + 10 * 20 = q * 20 + 200$.

Figure 7.1 shows the computed aggregate busy times. As the number of requests increases, the traditional worst-case based approach scales proportionally. In contrast, the aggregate approach only increases by the added core request time. This results from the fact that the possible interference in the complete time window is assumed independently of the number of requests. This is also the reason why this method is inferior for $q = 1$, as in traditional WCRT only the interference during the requests' actual busy window (not w) can interfere with the execution. As both approaches are conservative, the minimum can be taken for an optimal analysis result.

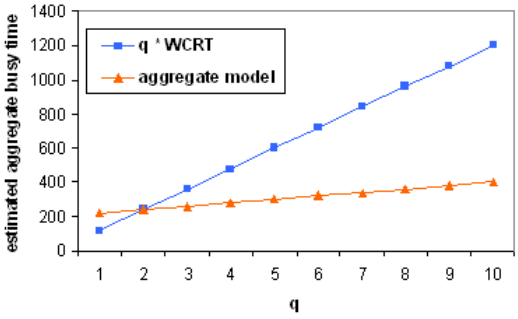

Figure 7.1: Aggregate Busy Time Analysis compared to Multiple Worst Case Scenarios.

To quantify the benefit of the aggregate busy time model for an actual system, we set up a synthetic example based on setup and simulation framework that is utilized and investigated in Section 8.2.3, where a platform configuration with 4 cores connected to a shared memory that is arbitrated first-come-first-served. One core executes a real-time task and the others perform latency insensitive image processing. Due to the common memory and interconnect, the computation on each core can not be considered independently. Rather, the current memory load from any of the cores impacts the run-time of tasks on the other cores.

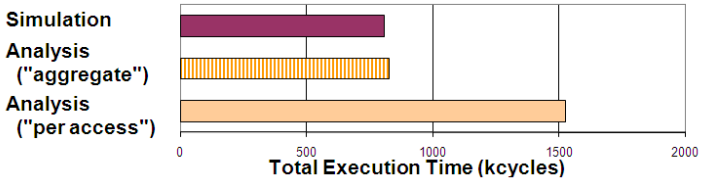

Figure 7.2: Formal Analysis Options compared to Measurements.

Assuming each processor thread can have only one open operation at a time, the worst case memory access time can be straight-forwardly bounded as the product of the number of processors and worst-case delay of each access. This time can be multiplied with the amount of memory accesses and added to the task's core execution time according to equation (7.2). The result of this method is depicted in Figure 7.2 (Analysis "per access"). If the same system is executed on the simulator, a much smaller response time is measured for the real time task (Simulation). The ca. 100% deviation shows the room for improvement. Repeating the analysis by resorting to the new analysis options, particularly the aggregate busy time (Analysis "aggregate"), delivers much tighter results in this setup

The benefit of resorting to the aggregate analysis approach as opposed to the multiple worst-cases has also been explored for larger task sets in [Sch06c] and [Sch08d]. The latter contains also a comparison of this approach to simulation based system investigation. Also as announced above, Chapter 8 provides experiments in which the analysis of the request latency is embedded into the analysis of a complete system.

7.7 Summary

In this chapter, we have investigated the total time that a shared resource may take to complete the processing of a set of requests. This metric, termed the *aggregate busy time*, is a contributor to the busy time of a task that requires external operations to complete its execution. Two methods have been provided: Firstly, the aggregate busy time is straight-forwardly bounded by the multiple of the worst-case time for an

individual request. But because accumulation of worst-cases may be very imprecise and thus preclude itself in practice, we have provided an new interface and analysis that focuses on the joint behavior of a set of requests in a larger time interval.

We have provided a generic analysis for shared resources that are arbitrated according to a work-considering arbiter, and a tailored analysis for round-robin arbitration. The set of covered arbiters can be easily extended through the modular approach.

To address more complex system setups we have provided a toolbox of methods that allow adapting the analyses correspondingly. This addresses systems in which the tasks make use of different target shared resources via buses or other independently arbitrated interconnects. We have also shown that due to its monotonic properties, the analysis can be integrated in the system level analysis approach presented in Chapter 2.

8 Applications

In this chapter, the analysis framework is applied to two challenging use cases. In the first setup, we adapt our analysis to consider the timing implications of using instruction caches to buffer the instructions fetched from a shared memory. In the second setup, a realistic multiprocessor system from STMicroelectronics is investigated.

8.1 The Timing of Multiprocessor Systems with Local Instruction Caches

The analysis of the shared resource delays provided in the previous Chapters 5 to 7 was based on the assumption that the requests are issued within known intervals during the execution of the tasks, and that these intervals are not distorted by the actual scheduling decisions. This model is accurate when the requests are the result of explicit instructions in the source code. However, when requests are the result of cache misses, their timing is the implicit effect of the current processor and system state and can heavily vary due to dynamic context switching.

As a countermeasure, static cache partitioning or run-time cache locking has been proposed to reduce the inter-task cache competition (see e.g. [Lie97, Cam01, Pua02]). While such strategies are highly useful to increase the system's predictability, they also imply an inefficient use of the available resources. Consequently, dynamically shared caches can be observed very commonly in practice. The scope of our analysis shall encompass setups in which no such measures are in place.

This challenging scenario can be addressed with the analysis proposed in the previous chapters and a three-step extension: In a first step, the number of "intrinsic" cache misses of an undisturbed task execution is bounded in Section 8.1.1. The second step presented in Section 8.1.2 derives the additional cache misses due to the run-time preemptions. This allows us to compute the number of cache misses during a tasks busy time in Section 8.1.2.3, and the processor's shared resource request bound in Section 8.1.2.4. We put these extensions into practice by investigating a benchmark application in Section 8.1.3.

8.1.1 Bounding Intrinsic Cache Misses

Bounding the tasks' intrinsic cache misses has been a heavily researched subject in the worst-case execution time community. By so-called abstract interpretation of the execution binary and an accompanying cache model, formal methods are able

to identify for each basic block the maximum number of cache misses that may occur during the execution [Wil08, Sta07]. This figure is conservative, but may be pessimistic because not all of these cache misses will actually happen during run-time. The task's control flow graph is then annotated with the possible cache miss delays per basic block, and the task's worst-case execution time is derived by searching for the longest path. This procedure is obviously only possible if the cache miss delay is constant, which is the case for single processor systems, but does not hold for shared memories with dynamic arbitration.

The existing methods can still be utilized for our purpose by means of a small adaptation. Instead of annotating the control flow graph with the actual *delay*, we annotate the basic blocks only with the bare *amount* of potential cache misses that go to a shared memory. This then enables the analysis of the maximum number of cache misses per task instance and minimum distance between cache misses with the methods discussed in Section 5.5.1 for explicit shared resource accesses. Let the result of this analysis be $N_{j\to M}^{intrinsic}$: the maximum number of cache misses that go to a shared memory M per instance of a task τ_j, and $\tilde{d}_{j\to M}^{intrinsic}(n)$: the minimum distance between these cache misses.

Once the number of implicit cache misses for each task is known, we can aggregate the load for a set of tasks T that is mapped to the same processor according to Theorem 5.4. Let the result of this analysis be denoted with $\tilde{\delta}_{T\to M}^{intrinsic}(n)$, representing the minimum distance between any n intrinsic cache misses.

8.1.2 Bounding Preemption-Related Cache Misses

When preemptive scheduling is involved, the preemption by a prioritized task can cause some useful cache blocks of a preempted task to be replaced, such that it may suffer additional cache misses when it resumes execution. This effect is called the *cache-related preemption delay* (CRPD) and is known from single processor systems, which has been investigated with formal analyses since [BM96].

8.1.2.1 Related Work on Cache-Related Preemption Delay

The approaches usually consist of analysis steps on two different levels: Firstly, the number of replaced cache blocks per preemption of a task pair is derived (as in [BM96, Lee01, Sta07, Alt09]). Secondly, the possible number of preemptions during scheduling is investigated (as in [Lee01, Pet01, Sta05b]). We now take a closer look at the existing solutions to these problems for these steps in order to build our analysis on it.

Without further investigating the tasks itself, the number of *replaced cache blocks per preemption* of a task pair is upper bounded by the total number of cache blocks in the cache [BM96]. Obviously, this is an overestimation if the tasks do not utilize the complete cache. A better bound can be obtained (still relatively easily) by computing

the minimum of the size of the set of cache blocks used by the preempting task and the size of the set of cache blocks used by the preempted task [BM96, Pet01].

More sophisticated analyses track the progressions of the cache state during the execution of each task. At any given point in the execution of a task, one can identify a set of "useful cache blocks", which is the set of cache blocks that have been loaded into the cache, and might be accessed again later when the execution continues. When this task is preempted, it will only be these useful cache blocks that trigger a cache-related preemption delay when the task resumes execution [Lee01]. Thus, the preemption cost is limited by the maximum number of cache blocks in the the intersection of the useful cache blocks of the preempted task and the used cache blocks of the preempting task. It has been shown that this metric delivers tighter analysis results [Lee01, Sta07], albeit at the cost of an increased analysis effort.

In a second analysis step, the *possible number of preemptions* that may occur during the response time of a task is investigated. In static priority preemptive scheduling the total number of preemptions corresponds to the number of task activations, which is bounded by the task activating event model. In our model, we assume that these event models are either known a priori (if they constitute time-triggered events or external interrupts) or they can be derived with our iterative approach in Chapter 3. By accounting the above cost for each preemption, one can derive a total cache related-preemption delay within the response time of a task [BM96, Lee01, Pet01]. The results can be improved by recognizing the fact that repeated preemptions by the same task only have a limited incremental effect on the cache state of the preempted tasks [Sta05b].

Due to the variable cache miss latency in multiprocessor systems, the above analyses are not directly applicable in our setup. However, we can adapt the techniques in order to extract the relevant information to re-enable our analysis approach. To facilitate the analysis, we rely on the metric of *preemption-related cache misses* (PRCM), which shall represent the bare number of cache misses (as opposed to the previously researched cache-related preemption delay that represents the time penalty).

8.1.2.2 Bounding PRCM per Preemption

First, we utilize the approach in [Sta07] to identify the preemption-related cache misses per task pair based on [Lee01] and [Neg03]. However, we will not factor in the cache miss penalty. The analysis is rather complex, and simpler versions such as suggested in [BM96] are also compatible. Let the result of this analysis be π^j: the maximum number of used cache blocks of a task τ_j, π_j: the maximum number of useful cache blocks of task τ_j, and π_i^j: the maximum number of useful cache lines of a task τ_i that may be replaced per preemption by a task τ_j. We can extrapolate

between these parameters as follows

$$\pi_i^j \leq \min\{\pi^j; \pi_i\} \tag{8.1}$$
$$\pi^j \leq \max_{i \in T}\{\pi_i^j\} \tag{8.2}$$
$$\pi_i \leq \max_{j \in T}\{\pi_i^j\} \tag{8.3}$$

Next, we investigate the number of preemptions during run-time. Fort his we consider two different analysis parameters: One the one hand, we are interested in the cache-related preemption delay during the busy time of a specific task τ; for this we need the number of preemptions and the resulting PRCMs in a time interval during which τ_i is not completed. On the other hand, we need the load imposed on the cache; in this scenario, all tasks on the processor can be involved.

8.1.2.3 Total Amount of PRCM during a Task's Busy Time

If a task τ_i is continuously ready in a time window of size w_i, this implies that no lower priority task receives the chance to execute once that τ_i has begun executing. Consequently, preemption-related cache misses can only be caused by preemptions of higher priority tasks. For each higher priority task, the maximum number of task activations is given by $\eta_j(w_i)$, and the upper bound on its inflicted cache eviction is given by π^j. Thus, we have the following total number of preemption-related cache misses during the τ_i's busy time w_i:

$$\Pi_i(w_i) = \sum_{j \in hp(i)} \eta_j(w_i)\pi^j \tag{8.4}$$

In addition, it is not uncommon that tasks are suspended during their execution for example due to critical sections (see also Section 6.3.4). This may lead to additional context switches that introduce two new types of opportunities for cache eviction: Firstly, each suspension by a high priority task causes a formerly preempted task to resume execution — and thus possibly reconstruct its cache state. Then when the high priority task resumes execution, it will again evict useful cache blocks. Secondly, when a task suspends, its own cache state can be disturbed by the low priority tasks that execute in between (this effect has been largely ignored in previous research such as [BM96] and [Lee01]).

Under PCP, the number of tasks that can execute once a higher priority task suspends due to a locked semaphore is bounded by one (i.e. the one task that is currently holding the semaphore). In multiprocessor setups, in which suspension-based synchronization protocols (such as MPCP [Raj88]) are applied, more than one local task may execute while a high priority task is suspended. Let $lp(i)$ be the set of

tasks that may execute during a suspension of task τ_i, and v_j be the number of such voluntary suspensions per task τ_j, then we have

$$\Pi_i(w_i) = \sum_{j \in hp(i)} \left((v_j + 1) \cdot \eta_j(w_i) \cdot \pi^j \right) + v_i \sum_{l \in lp(i)} \pi^l \tag{8.5}$$

Further solutions to this problem are provided in [Pet01] and [Sta05b] for static priority preemptive scheduling with different trade-offs between complexity and accuracy. These algorithms can also be applied, albeit on the basis of the bare number of cache misses instead of the cache miss delay.

Now we can derive the total number of cache lines that are evicted due to scheduling in the busy window of a task τ_i. This value is relevant in the computation of the task's response time e.g. according to Theorem 6.1. Next, we will apply a similar reasoning in order to bound the total load imposed a the memory from a set of tasks in an arbitrary time window.

8.1.2.4 Overall Amount of PRCM in an Arbitrary Time Window

For bounding the total load imposed by a set of tasks T in an arbitrary time window, the analysis has to encompass all tasks in T, and the combination of their possible run-time scheduling and not only the possible traces that are observed during the busy time of a specific task as derived in the previous section. Section 8.1.1 has already provided the maximum number of intrinsic cache misses of such a set of tasks. In addition, we will now bound the worst-case amount of preemption-related cache misses in an arbitrary time window of size w.

In an investigated time window, the overall amount of context-switch related cache misses is given by the sum over a) the cache blocks evicted by each preemption (i.e. task activation or resumption after a suspension) — this is bounded by π^j per activation of each task; and b) the cache blocks evicted by lower priority tasks while a task is suspended — this is bounded by the maximum amount of useful cache lines π_j of each suspending task (or more accurately: the amount of useful cache lines just before a suspension). This reasoning leads us to the following bound:

$$\Pi_{T \to M}(w) = \sum_{j \in T} \left((v_j + 1) \cdot \eta_j(w) \cdot \pi^j \right) + \sum_{j \in T} (v_j \cdot \pi_j) \tag{8.6}$$

Equation (8.6) provides the total number of preemption-related cache misses of all tasks T on a processor within a given time window w. Depending on the cache configuration, this bound could be improved by relying on specific maximal intersections of useful and used cache blocks of the involved tasks, but this would come at the cost of a significantly increased analysis complexity both for deriving the preemption data per task pair and the investigation of worst-case combinations for the task set

(a starting point is again [Sta05b], where this has been exploited for non-suspending tasks).

With this the total number of cache misses consisting of the *intrinsic cache misses* and *preemption-related cache misses* can now be considered in the computation of the load imposed by a set of tasks T on a shared memory M.

The preemption-related cache misses cause the tasks to produce more requests in shorter time intervals. The resulting minimum distance between any n cache misses that go to a shared memory M can be computed using the bounds on the intrinsic cache misses $\tilde{\delta}^{intrinsic}_{T\to M}(n)$ and the additional scheduling-related cache misses $\Pi_{T\to M}(w)$ on a processor within a time interval w:

$$\tilde{\delta}^{cache}_{T\to M}(n) = \min[0, \tilde{\delta}^{intrinsic}_{T\to M}(n - \Pi_{T\to M}(w))] \tag{8.7}$$

The distance between the cache misses in Equation 8.7 represents the relevant shared resource request bound to be used for the conflict analysis on the shared memory according to Section 7.1.1. It will be iteratively refined for different time windows of size w during the computation of a task's response time or the analysis of another timing parameter as provided in Chapter 2.

8.1.3 Experimental Evaluation

To evaluate the proposed method, we integrate the shared resource request bounds derived above into the worst-case response time analysis for tasks in multiprocessor systems with shared resources (following Chapters 3 to 7). For each task τ_i, its response time is computed according to Theorem 6.1, which includes a) its own worst-case execution time, b) the possible preemption time due to higher priority tasks, and c) the delays experienced when waiting for cache misses to be serviced. The latter is computed according to Theorem 7.1 as the sum over the duration of τ_i's requests and the interfering requests of other tasks mapped to the same processor (computed according to Section 8.1.2.3), and the requests by tasks on other processors in the same time window (computed according to Section 8.1.2.4).

We consider a system with 2 processors that are equipped with local instruction caches (L1) and which can access a shared memory as depicted in Figure 8.1. All data is stored in local memories. When an (instruction) cache miss occurs, the corresponding processor is stalled until the request has been served by the memory. The memory serves requests first-come-first-served with a constant time of 5 time units per access (i.e. there is no L2 cache). Each processor is running two tasks parametrized according to Table 8.1. Other than the shared memory, the tasks do not access any shared resources. The tasks are actual benchmarks taken from [Sta07] investigated with a modified version of the worst-case execution time analysis tool [Sta05a]. The derived WCET values without cache miss delays are listed in Table 8.1.

A different number of intrinsic cache misses and preemption-related cache misses is obtained for various cache configurations (cache size between 64 and 1024 Byte;

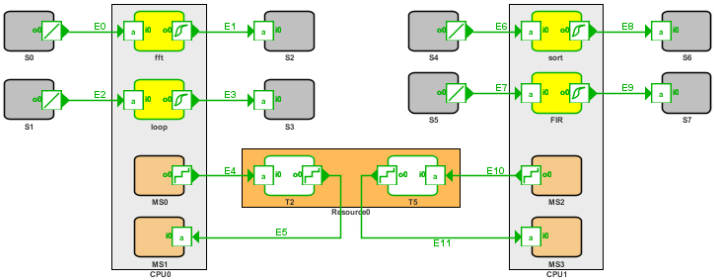

Figure 8.1: Experimental Setup.

Task	Processor	Priority	Period	WCET	$N_{j\to M}^{max}$ [64,128,256,512,1024]B
countsort	CPU0	1	20000	168	[55,98,80,12,60]
whetstone	CPU0	2	75000	57253	[790,790,790,550,50]
FIR	CPU1	1	20000	2083	[155,110,8,35,8]
exchangesort	CPU1	2	40000	11011	[1115,710,710,710,710]

Table 8.1: Experimental Setup.

direct mapped; replacement strategy least-recently-used). It can be observed that the development is not monotonic, but that there is a general tendency that the intrinsic cache misses decrease with larger cache sizes. A different tendency can be observed for the preemption-related cache misses for which the cost of a single preemption is listed in Table 8.2. The PRCMs increase with growing cache size, because more useful cache blocks can be replaced upon a preemption. When the cache is sufficiently large, the displacement is again reduced.

These values are now used to derive the load imposed from each processor to the shared memory according to Equation 8.7, and the delays and response times are computed as described in the first paragraph of this section.

Figure 8.2 shows the resulting response times of the *whetstone* task for the different

Preemption Scenario	64B	128B	256B	512B	1024B
"*countsort* preempts *whetstone*"	8	15	32	46	25
"*FIR* preempts *exchangesort*"	8	15	31	63	15

Table 8.2: Premption-Related Cache Misses (π_i^j) for different Cache Configurations.

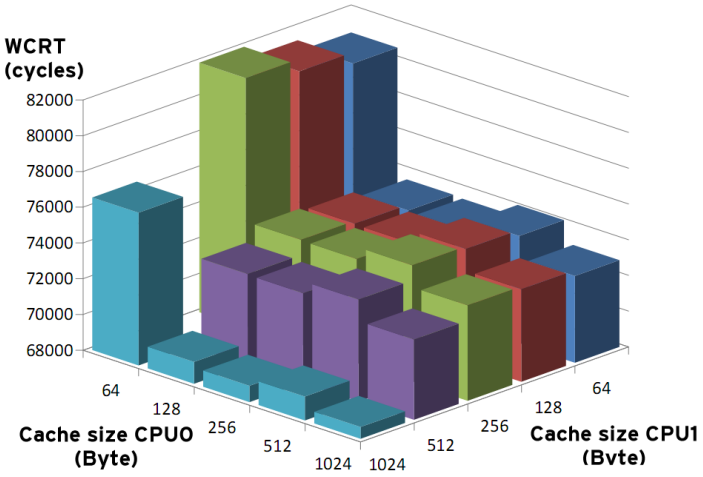

Figure 8.2: Resulting Worst-Case Response Times (WCRT) for *whetstone* task on CPU0 [Sch10b].

cache configurations. As can be expected, the response time is influenced by both the local cache configurations and that of the other processor, as either misses will cause an execution delay. The result is not monotonic with respect to cache size, as the intrinsic and the preemption-related cache misses develop in different directions. One configuration has lead to a response time that was beyond the task deadline (64B cache on CPU0, and 512B cache on CPU1). The most economical configuration is a cache size of 128B on CPU0 and 64B on CPU1, as larger cache sizes do not deliver a dramatically improved performance.

8.1.4 Conclusion

In this section, the suggested analysis framework was applied to a multiprocessor system with local caches that are connected to a shared memory. Because the timing of the cache misses is dynamic and hard to predict, it is highly challenging to deliver real-time guarantees in such a setup. We have shown how our compositional analysis approach can be adapted to cover also this scenario. For this, we have drawn from previous research in the domain of single-processor cache analysis and developed new analysis components to bound the dynamic load on the shared memory. This has enabled us to verify a hypothetical setup with actual benchmarks and very quickly explore a set of different cache configurations.

8.2 Performance Analysis of the StepNP Multiprocessor Platform

The StepNP platform [Pau02] has been introduced by the STMicroelectronics advanced system technology organization as an experimental MpSoC target platform for the MultiFlex platform mapping tools [Pau06]. It is general-purpose, but can be adopted to suit the demands of various application domains. StepNP is not used in a commercial product, but it has served as a baseline to support the exploration of platform mapping tools for next generation platforms (such as Nomadik [Pag07]) The StepNP platform is still very interesting for investigation, as it represents a realistic system. A number of applications have already been ported to the platform [Bou06][Pau06] to investigate application behavior and tune architecture design decisions.

8.2.1 Platform Architecture

The basic StepNP platform consists of a set of fully programmable RISC processors and a standardized interconnect. Figure 8.3 shows the three basic components of the platform: processor engines (in this case 4 RISC based processors), an interconnect (the STBus communication infrastructure), and some specialized coprocessors (in this case, two hardware-based scheduling engines with additional hardware units to support SMP and message-passing programming models [Pau06]).

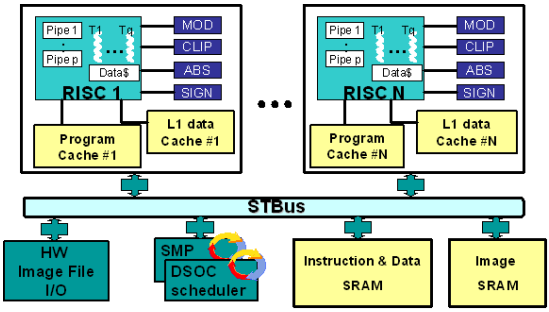

Figure 8.3: StepNP Base Platform.

As introduced in Chapter 1, memory access latency is an issue of growing concern in any embedded system design. In the given platform concept the processors are therefore equipped with hardware multithreading capability. This allows effective latency "hiding" where CPU cycles are not wasted but can be used by other threads. Such a hardware multithreaded processor has a separate register bank for each thread, allowing low-overhead context switching between threads, often with no disruption to the processor pipeline [Pau02].

In the original concept, a crossbar and a multi-bank memory are used to deliver orthogonal performance to each processor. In the application provided in this section, we rely on load models derived from per-task simulations to analytically bound the impact of shared interconnect on timing. One result is that in the example application, a single shared bus would also deliver sufficient performance.

8.2.2 Image Processing Application

The example application chosen for this investigation was selected and provided by the École Polytechnique de Montréal and has been mapped to the StepNP platform. It is an image processing algorithm for video applications that consists of 5 successive filtering and processing steps (see Figure 8.4). Each of these 5 application functions fetches the resulting image produced by the predecessor from the cache or implicitly from the shared memory, performs its necessary operations (mostly on the cached data), and leaves the result in the shared memory for the next stage. The frames are processed sequentially. Each processing step can be parallelized into $n = 2^x$ independent tasks, where x is configurable. The parallelization represents a spacial dissection of the original frame into equally sized tiles. When a new frame has arrived at the system's input, the task is forked into n subtasks that are assigned to the available threads. After all subtasks have completed execution the image is merged again for the next step.

For efficiency reasons, no software multiplexing is implemented, so that the number of forked subtasks is bounded by the number of available hardware threads (number of CPUs multiplied with the number of threads per CPU). The forking and merging is controlled by a user thread running on one of the CPUs in between the pipeline functions. All memory operations pass via the same interconnect to the same memory (see Sec. 8.2.1).

8.2.3 Experiments

For the following study of the complete system described in above, we adopt a mixed methodology. We use the available timing aware simulators to investigate the timing of individual components (i.e. tasks) in a reasonable amount of time. This removes the need to derive specific models of the tasks and their execution environment which would be required for a formal per-task analysis. Nevertheless, formal methods such as reviewed in [Wil08] and Section 8.1 can be used to achieve higher confidence in the extracted task timing and consequently the overall analysis results (and should be used for safety-critical applications).

Based on the timing information per-task, we apply the formal analysis framework presented in this thesis (in particular Chapters 5 to 7) to quickly and reliably derive the integration effects on the system level with robust accuracy.

We collected the data in isolated simulations of each application function. A simulation run can yield the following results between two breakpoints: Total execution

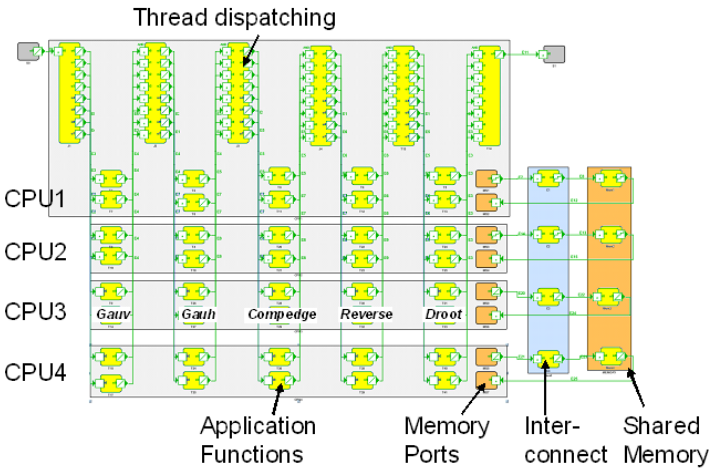

Figure 8.4: Image Processing Application.

time, number of cache hits, number of cache misses, number of writes. By taking care that no other tasks are active in the system, these values can directly be attributed to one task. In our case study we use a benchmark input image for this purpose. This was relatively accurate as the nature of the algorithm is such that it shows no input data dependent behavior. The cache offers single cycle access to the active thread, so that we consider the cache hit delay as part of the execution time. A cache miss will incur a waiting time for the requesting task that consists of the request latency via the bus plus the access time to the memory. Although the delays are actually input parameters to the simulator, we have independently determined them through measurements.

8.2.3.1 Single-threaded Setup

In the first experiment, we assume that the application is divided into 4 subtasks, each of which is mapped to its own core (i.e. we have no multithreading), and the shared memory can be accessed from each core without inter-core interference. Figure 8.5 shows the resulting response time for each of the 5 application functions. The first bar ("simulation (4x1Thr)") represents the simulated execution time if a dissected input image is concurrently processed by the four CPUs. Next, we performed our formal analysis with the data previously gathered from the exclusive function simulation (second bar, "Analysis Crossbar"). As there are no additional conflicts on processors, crossbar, or the memory, we receive very accurate results that closely resemble the

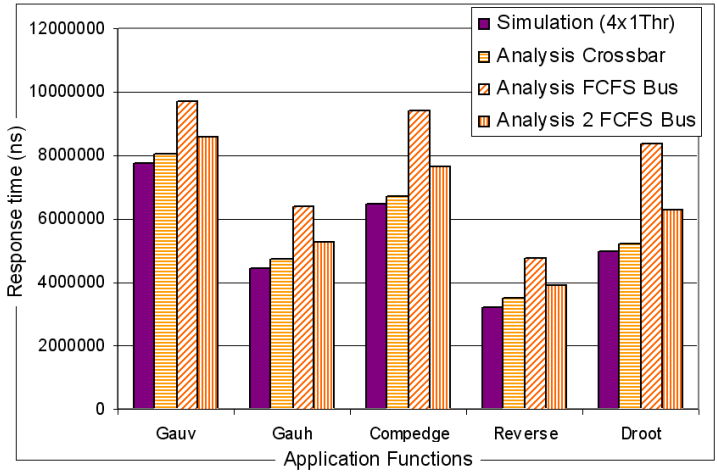

Figure 8.5: Experiments for Singlethreaded Setup.

simulation.

Now we modify the model of the bus and the memory to exclusively treat one request at a time in a first-come-first-served ordering. This is easily introduced into the analysis of each task by including the memory interference in the tasks' aggregate busy times provided in Theorem 7.1. The conservative model of the interference will now contain all memory accesses by the tasks that are active at the same time. The third bar ("Analysis FCFS Bus") in Figure 8.5 shows the predicted response time for each application function. The response times of the functions are affected by the contention on the bus and memory to different degrees. Depending on the amount of memory traffic the response times increase by 25% for *Gauv* and up to 41% for *Droot*.

The end-to-end latency in this example is given by the sum of the worst-case response times of each application function. This is captured by the algorithm presented in Chapter 4, which recursively finds the longest "passage" through the application topology (in this case, all such passages actually have the same delay in the worst case). Note that previous algorithms with predominantly local focus produce far larger latencies, in particular because the synchronization information at the merge operation before each application function is lost, which leads to a large assumed synchronization delay ("AND-activation delay" [Jer05]).

In a concluding option we assume a hypothetical memory and bus controller that allows two parallel accesses which reduces the interference by half (4th bar, "Analysis

2 FCFS bus"). A designer can now choose the cheapest bus structure that is still guaranteed to deliver sufficient performance.

8.2.3.2 Multithreaded Setup

The second series of experiments assumes each application function is parallelized into 8 subtasks. Again, we derived single subtask behavior by simulation in isolation. The first two bars ("Simulation (8x1Thr)" and "Analysis Crossbar (8x1Thr)") in Figure 8.6 show that our approach can again precisely capture the actual behavior for 8 concurrent subtasks on 8 cores.

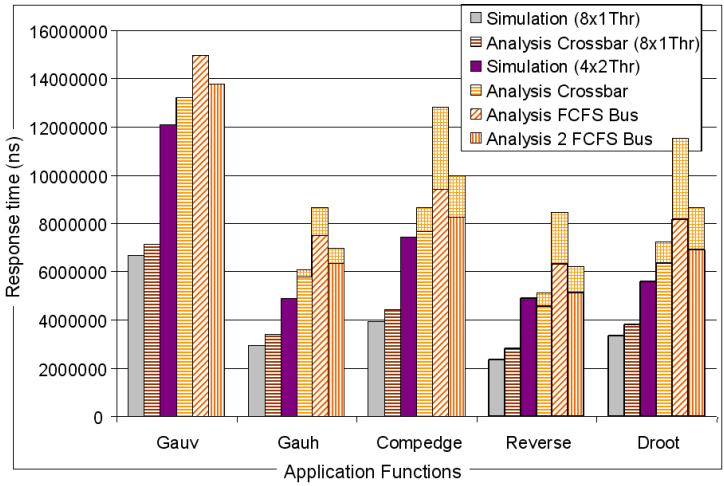

Figure 8.6: Experiments for Multithreaded Setup.

We then assume that two subtasks are mapped to hardware threads on the same processor. This will cause competition for the processor, and also for the cache content. The third bar ("Simulation (4x2Thr)") shows that for most functions (*Gauv*, *Gauh*, *Compedge*, and *Droot*) processor sharing increases the measured response time by less than 100%. This can be attributed to how efficiently the memory accesses interleave during run-time. However, the measured response time for *Reverse* is more than twice as large: By mapping two tasks to the same core, the required execution time will remain unaffected, but the cache miss rate may increase due to cache thrashing.

For conservative results, this behavior has to be covered by the analysis. For the scope of this experiment, we have measured the additional cache misses for each function observed under dualthreaded simulations. In general, simulation is unreliable to find

worst-case cache misses due to the large space of possible application and cache states, please refer to Section 8.1 for safe analysis results based on formal cache analysis. In the given setup however, the state space is much smaller, because a) the input data does not impact the number of cache misses and b) the thread-offsets vary only insignificantly due to the fork-join structure of the application. The contribution of effect to the response time is shown in the respective upper parts of each column.

The functions' response times are derived according to Theorem 6.4 for the local round-robin thread scheduling, and Theorem 7.1 for the arbitration of the shared bus and memory. Also for the setup with 4 dual-threaded processors, we explore the option of utilizing a crossbar or a shared first-come first-served bus which allow only one or two simultaneous transactions. Functions which perform more memory accesses (*Compedge*, *Reverse*, or *Droot*) again suffer more severely from the resulting bus competition (as can be seen in the last two bars). The overall analysis speed in this approach is very high when compared to purely simulation-based investigation. Each simulation run of the *individual* task functions already took minutes to complete, which becomes a severe problem if system level options were investigated with this technique. By contrast, each analysis result was calculated in less than a second due to the abstraction from the actual functionality.

8.2.4 Summary

In this experiment, we applied our formal performance analysis methodology to a realistic embedded multiprocessor system on chip. This was possible by addressing and quantifying the impact of the complex interdependencies that surface when shared memories are used. As opposed to the procedure in Section 8.1, we adopted an engineering approach to deriving the input data for our system level analysis: Per task, we extracted the relevant parameters by simulation. The local task interaction in the multithreaded round-robin scheduler was captured in our analysis allowing the prediction of the worst case response times. The memory accesses are analysed with high speed and precision by relying on the concept of aggregate busy times instead of deriving individual request timing. The approach was used to gather worst-case performance metrics and quickly derive accurate estimates for various interconnect options.

9 Conclusion

The embedded systems industry continues to introduce product innovations in order to meet the persistent consumer demand for more safety, more personal and work productivity, and more comfort at a competitive cost. But these innovations entail a growing complexity of hardware and software architectures that poses challenges to the safe design and application of such systems. These challenges are aggravated by technological constraints such as the power wall that prohibit simple scaling of existing solutions. It is therefore crucial that the designers are equipped with methods and tools to handle an increasing design complexity, improve the productivity in the development process, and safeguard the release of a product. Today, a new generation of tool vendors is providing solutions to performance analysis and verification. These solutions are applied and evaluated in the automotive industry, and other domains such as aerospace, automation, and medical applications in different phases of the design process.

But the application of the necessary formalized approaches is still too often constrained to small systems or isolated subsystems. This is to a large extend due to several concerns that are identified and ameliorated in this thesis as will be highlighted below: For example, a general concern is often voiced about the *accuracy* of the derived metrics. For this, we revisited key performance metrics to provide more accurate predictions. Then the industry needs solutions that provide *suitable models* which respect their specific solutions and allow capturing a possibly large spectrum of actual design patterns. This thesis generalizes the set of investigated performance metrics and provides new modeling capability for multicore setups. Finally, formal methods can only be successfully introduced into the existing development process, if they can be expected to allow the *future adaptation* to architectural and methodological changes. Only compositional analysis approaches, such as the one provided in this thesis, can meet these expectations.

Far-reaching decisions in the component dimensioning of embedded multiprocessor systems are based on the results of performance analyses, yet in some setups key metrics were still suffering from significant overestimation. As a principal performance indicator, an analysis is expected to deliver for each task in a system its worst-case response time and, in setups where the processing involves more than one component, the end-to-end latencies counted from the arrival of an external event until it has been completely processed and the output is generated. The methods provided in

this thesis enable the computation of this latency with supreme accuracy by explicitly considering the transience of overload situations. Still, when compared to alternative methods, the analysis remains efficient by investigating only a small set of relevant analysis candidates. Through similar reasoning, the workload imposed by individual tasks on its processing unit is now covered with greater precision. The general load event model interface utilized in this thesis allows capturing dynamic and complex event patterns, and the generalized event model propagation technique allows their derivation throughout the system. Previous limitations, such as the concentration on a specific set of standard event models could be removed. We showed that this provides better workload approximations, in particular in the presence of chained task activations, which indirectly leads to better worst-case response time bounds.

While symmetrical multicores have already been applied in high-performance and power-sensitive embedded domains, there is a growing interest in applying these architectures also in control-dominated hard real-time systems. Also here, the increased demand for performance, power-efficiency, and reliability make this move inevitable. But the previous modeling capabilities were inadequate to provide worst-case guarantees in the presence of tight inter-processor timing dependencies and dynamic shared resource arbitration, which thwart the traditional bottom-up analysis approaches. For this reason, this thesis has introduced new models and analyses for multiprocessor systems with globally shared resources. Most notably, a model was introduced to efficiently and conservatively capture the timing of even large amounts of shared resource operations, and to consider the resulting latency in the local analyses across all involved processor cores.

The industrial as well as academic requirement of adaptability to future modifications to the applied hardware and software architectures has led to the adoption of a compositional analysis approach. Previous semi-formal work has been formalized and generalized in this thesis: The proposed framework drops the predominant focus on task activating event models and allows the iterative analysis of a diverse, heterogeneous set of interdependent analysis parameters. Still, it remains compatible to established compositional analysis procedures. The analysis of the multicore setup previously mentioned was the first application that could benefit from this generalization. The requirements to extend the set of investigated parameters in the future have been clearly identified.

The methodology was implemented into a research version of “SymTA/S”, an analysis tool originally developed at the Technische Universität Braunschweig based on previous work on compositional performance analysis as explained in this thesis, that is now available and developed also as a commercial framework. For the scope of this thesis the approach was put into practice in two setups: In one, a multiprocessor system with shared memory and local caches was investigated. By combining methods from the single-processor cache analysis with the proposed system-level approach, this is the first time that hard real-time guarantees could be supplied for such a setup. In

another application, a realistic industrial multiprocessor platform with multithreaded scheduling was investigated with a hybrid approach that integrated the measurements from isolated single-task simulation into the formal system-level analysis. In addition to the application and evaluation of the complete analysis framework, also the individual analysis functions have each been benchmarked against existing solutions in representative examples, where they have delivered respectable improvements.

A promising direction of future research efforts is to reconcile analytical performance analysis approaches with the efforts to orthogonalize system resources in multiprocessor systems. Generally, a setup can be made more predictable by reducing the state space and component inter-dependencies for example through time-driven scheduling or budgeting. But such techniques can usually not be universally applied due to performance concerns, incomplete hardware support, and global data sharing. Thus, a combination of the techniques to reduce the system's state space and dedicated analysis of the remaining dynamic behavior based on the algorithms suggested in this thesis could prove to provide the best solution with respect to design robustness and adaptivity, cost, and provable performance.

As noted above, the analysis framework is conceived to be extensible to other combinations of scheduling and resource arbitration policies. Currently, a major industrial concern is the definition of multiprocessor synchronization protocols that meet practical requirements. At this phase, the proposed analysis framework can be very instrumental in guiding the decisions, because it can provide generic performance indicators for a large array of different setups — even before the specification of the corresponding software, operating system, and hardware has been completed.

List of Publications

Journal Articles and Book Chapters

[1] Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Transactions on Embedded Computing Systems (Special Issue on Model Driven Embedded System Design)*, 2010. (to appear).

[2] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Response time analysis in multicore ecus with shared resources. *IEEE Transactions on Industrial Informatics*, 5(4), November 2009.

[3] Simon Schliecker, Jonas Rox, Mircea Negrean, Kai Richter, Marek Jersak, and Rolf Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):979–992, July 2009.

[4] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Journal Design Automation for Embedded Systems (available as online first, April 2008)*, 13(1):27–49, June 2009.

[5] Simon Schliecker, Jonas Rox, Rafik Henia, Razvan Racu, Arne Hamann, and Rolf Ernst. Formal performance analysis for real-time heterogeneous embedded systems. In Gabriela Nicolescu and Pieter J Mosterman, editors, *Model-Based Design of Heterogeneous Embedded Systems*, tational Analysis, Synthesis, and Design of Dynamic Systems, chapter 3, pages 57–92. CRC Press, November 2009.

Publications at Conference and Workshops

[6] Mircea Negrean, Simon Schliecker, and Rolf Ernst. Timing implications of sharing resources in multicore real-time automotive systems. In *SAE World Congress*, volume System Level Architecture Design Tools and Methods (AE318), Detroit, MI, USA, April 2010. SAE International.

[7] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proc. of Design, Automation, and Test in Europe (DATE)*, Dresden, Germany, March 2010.

[8] Simon Schliecker and Rolf Ernst. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *Proc. 7th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Grenoble, France, October 2009. ACM.

[9] Mircea Negrean, Simon Schliecker, and Rolf Ernst. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *Proc. of Design, Automation, and Test in Europe (DATE)*, Nice, France, April 2009.

[10] Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Proc. 6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Atlanta, GA, October 2008.

[11] Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre Paulin, and Rolf Ernst. Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proc. 6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Atlanta, GA, October 2008.

[12] Simon Schliecker, Arne Hamann, Razvan Racu, and Rolf Ernst. Formal methods for system level performance analysis and optimization. In *Proc. of the Design Verification Conference (DVCon)*, San José, CA, February 2008.

[13] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proc. ACM Conference on Embedded Software (EMSOFT)*, Salzburg, Austria, October 2007.

[14] Simon Schliecker, Steffen Stein, and Rolf Ernst. Performance analysis of complex systems by integration of dataflow graphs and compositional performance analysis. In *Proc. of Design, Automation and Test in Europe (DATE)*, April 2007.

[15] Simon Schliecker, Matthias Ivers, and Rolf Ernst. Integrated analysis of communicating tasks in mpsocs. In *Proc. 3rd International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Seoul, Korea, October 2006.

[16] Simon Schliecker, Matthias Ivers, and Rolf Ernst. Memory access patterns for the analysis of mpsocs. In *North-East Workshop on Circuits and Systems*, Gatineau, Canada, June 2006. IEEE, IEEE.

[17] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Palma de Mallorca, Spain, July 2005.

[18] Simon Schliecker, Steffen Stein, Joern-Christian Braam, and Martin Schnieringer. System level performance analysis with formal methods and virtual prototyping, embedded world conference, Nürnberg, Germany. Embedded World Conference, February 2008.

[19] Simon Schliecker, Matthias Ivers, Jan Staschulat, and Rolf Ernst. A framework for the busy time calculation of multiple correlated events. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, number 06902, Dresden, Germany, July 2006.

[20] Jan Staschulat, Simon Schliecker, Matthias Ivers, and Rolf Ernst. Analysis of memory latencies in multi-processor systems. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Palma de Mallorca, Spain, July 2005.

Other Publications

[21] Simon Schliecker, Matthias Ivers, and Rolf Ernst. A proof for memory access patterns for the analysis of mpsocs. Technical Report IDA-2006-01, Institute for Computer and Communication Network Engineering, Braunschweig, Germany, April 2006.

[22] Steffen Stein, Jonas Diemer, Matthias Ivers, Simon Schliecker, and Rolf Ernst. On the convergence of the symta/s analysis. Technical Report, Technische Universität Braunschweig, Braunschweig, Germany, November 2008.

Bibliography

[Adi02] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Network Processors*, volume 6(3), 2002.

[Aer03] Aeronautical Radio, Incorporated (ARINC). ARINC 653-1 Avionics Application Software Standard Interface. Standard, introduction here: `http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf` (retreived 2010-08-26), October 2003.

[Alb06] K. Albers, F. Bodmann, and F. Slomka. Hierarchical Event Streams and Event Dependency Graphs: A New Computational Model for Embedded Real-Time Systems. In *Proc. of the 18th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 97–106. IEEE Computer Society Washington, DC, USA, 2006.

[Alt09] S. Altmeyer and C. Burguière. A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay. In *Proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 109–118. IEEE Computer Society, July 2009.

[Alu90] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443, p. 322. Springer Verlag, Warwick University, England, July 1990.

[And01] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pp. 193–202, Dec. 2001.

[And03] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 33–40, July 2003.

[And08] A. Andrei, P. Eles, Z. Peng, and J. Rosen. Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *VLSI Design*, pp. 103–110, 2008.

[Ang08] J. Angermeier, U. Batzer, M. Majer, J. Teich, C. Claus, and W. Stechele. Reconfigurable HW/SW Architecture of a Real-Time Driver Assistance System. In *Proceedings of the 4th international workshop on Reconfigurable*

Computing: Architectures, Tools and Applications, p. 159. Springer-Verlag, 2008.

[ARM09] ARM Ltd. The ARM Cortex-A9 Processor, rev 2.0. ARM white paper, `http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf`, retrieved 2010-06-23, September 2009.

[aut06] AUTOSAR Partnership. `http://www.autosar.org/` (retrieved 2010-06-30), 2006.

[AUT09] AUTOSAR Partnership. Specification of Multi-Core OS Architecture V. 1.0.0 of Release 4.0, 2009.

[Bac92] F. Bacelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. John Wiley & Sons, Inc., 1992.

[Bai08] C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Bak05] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time. Technical report, Florida State University, July 2005.

[Bar07] S. K. Baruah and N. W. Fisher. The partitioned dynamic-priority scheduling of sporadic task systems. *Real-Time Systems*, volume 36(3):pp. 199–226, 2007.

[Bar08a] J. Barre, C. Rochange, and P. Sainrat. A Predictable Simultaneous Multithreading Scheme for Hard Real-Time. *Lecture Notes in Computer Science*, volume 4934:p. 161, 2008.

[Bar08b] S. Baruah and T. Baker. Global EDF Schedulability Analysis of Arbitrary Sporadic Task Systems. *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 3–12, July 2008.

[Bec93] M. Becker, M. Allen, C. Moore, J. Muhich, and D. Tuttle. The Power PC 601 Microprocessor. *IEEE Micro*, volume 13:pp. 54–68, 1993.

[Beh07] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharingin real-time open systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pp. 279–288. ACM, New York, NY, USA, 2007.

[Bek04] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen. Predictable embedded multiprocessor system design. In *Proc. of the SCOPES workshop*. Springer, 2004.

[Ben91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, volume 79(9):pp. 1270–1282, 1991.

[Ben02] L. Benini and G. De Micheli. Networks on chips: A new SoC paradigm. *Computer*, volume 35(1):pp. 70–78, 2002.

[Ben04] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling: capturing causality and the correctness of loosely time-triggered architectures (LTTA). In *Proceedings of the 4th ACM international conference on Embedded software*, p. 229. ACM, 2004.

[Ben07] A. Benveniste, P. Caspi, M. di Natale, C. Pinello, A. Sangiovanni-Vincentelli, and S. Tripakis. Loosely time-triggered architectures based on communication-by-sampling. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pp. 231–239. ACM, New York, NY, USA, 2007.

[Ben08] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *ACM Trans. Embed. Comput. Syst.*, volume 7(4):pp. 1–36, 2008.

[Ber07] M. Bertogna and M. Cirinei. Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms. In *Proc. 28th IEEE Real-Time Systems Symposium (RTSS)*, pp. 149–160, December 2007.

[Bet92] R. Bettati and J. Liu. End-to-end scheduling to meet deadlines in distributed systems. *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 452–459, 1992.

[Bil96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on signal processing*, volume 44(2):pp. 397–408, 1996.

[Ble05] K. Bletsas and N. Audsley. Extended analysis with reduced pessimism for systems with limited parallelism. In *Proc. of the 11th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 525–531, 2005.

[Blo07] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A Flexible Real-Time Locking Protocol for Multiprocessors. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 47–56. IEEE Computer Society, Washington, DC, USA, 2007.

[BM96] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proc. of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS)*. IEEE Computer Society Washington, DC, USA, 1996.

[Bor05] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, volume 25(6):pp. 10–16, 2005.

[Bou49] N. Bourbaki. Sur le théorème de Zorn. *Archiv der Mathematik*, volume 2(6):p. 434–437, 1949.

[Bou06] Y. Bouchebaba, G. Nicolescu, E. Aboulhamid, and F. Coelho. Buffer and register allocation for memory space optimization. *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)-Volume 00*, pp. 283–290, 2006.

[Bou09] A. Bouillard, L. T. X. Phan, and S. Chakraborty. Lightweight modeling of complex state dependencies in stream processing systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 195–204. San Francisco, CA, USA, 2009.

[Bra08a] B. Brandenburg and J. Anderson. A Comparison of the M-PCP, D-PCP, and FMLP on LITMUS RT. *Principles of Distributed Systems*, pp. 105–124, 2008.

[Bra08b] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 342–353, April 2008.

[Bre08] A. Brekling, M. Hansen, and J. Madsen. Models and formal verification of multiprocessor system-on-chips. *Journal of Logic and Algebraic Programming*, volume 77(1-2):pp. 1–19, 2008.

[Bro07] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, volume 95(2):pp. 356–373, 2007.

[Bun07] Bundesministerium für Umwelt Naturschutz und Reaktorsicherheit (BMU). Verkehr und Umwelt - Herausforderungen. `http://www.bmu.de/verkehr/herausforderung_verkehr_umwelt/doc/40203.php` (retrieved 5-5-2010), Berlin, Germany, September 2007.

[Bur95] A. Burns and A. Wellings. Engineering a hard real-time system: From theory to practice. *Software: Practice and Experience*, volume 25(7):pp. 705–726, jan 1995.

[Cam01] M. Campoy, A. P. Ivars, and J. V. B. Mataix. Static Use of Locking Caches in Multitask Preemptive Real-Time Systems. In *In Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop*, 2001.

[Cha92] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, volume 27(4):pp. 473–484, 1992.

[Cha03a] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. *Proc. 6th Design, Automation and Test in Europe (DATE)*, pp. 190–195, 2003.

[Cha03b] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, volume 41(5):pp. 641–665, 2003.

[Cha05a] S. Chakraborty, L. T. X. Phan, and P. S. Thiagarajan. Event count automata: A state-based model for stream processing systems. In *Proc. of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pp. 87–98. Miami, Florida, dec 2005.

[Cha05b] S. Chakraborty and L. Thiele. A New Task Model for Streaming Applications and Its Schedulability Analysis. *Proceedings of the Design, Automation and Test in Europe (DATE'05) Volume 1-Volume 01*, pp. 486–491, 2005.

[Col03] A. Colin and S. Petters. Experimental evaluation of code properties for WCET analysis. In *Real-Time Systems Symposium*, pp. 190 – 199, 3-5 2003.

[Cro03] P. Crowley and J. Baer. Worst-Case Execution Time Estimation for Hardware-assisted Multithreaded Processors. *Proc. of the 2nd Workshop on Network Processors*, pp. 36–47, 2003.

[Cru91a] R. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on Information Theory*, volume 37(1):pp. 114–131, 1991.

[Cru91b] R. Cruz. A calculus for network delay, part II: Network analysis. *IEEE Transactions on Information Theory*, volume 37(1):pp. 132–141, 1991.

[Cul10] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[Cur00] C. Curio, J. Edelbrunner, T. Kalinke, C. Tzomakas, and W. Von Seelen. Walking pedestrian recognition. *IEEE Transactions on intelligent transportation systems*, volume 1(3):pp. 155–163, 2000.

[Dan04] J. Dannenberg and C. Kleinhans. The coming age of collaboration in the automotive industry. *Mercer Management Journal*, volume 17:pp. 88–94, 2004.

[Das04] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, volume 9(4):pp. 385–418, 2004.

[Dav90] B. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Dav09] A. David, J. Illum, K. Larsen, A. Skou, and A. Univeristy. Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. *Model-Based Design for Embedded Systems*, p. 93, 2009.

[Drö98] W. Dröschel, W. Heuser, and R. Midderhoff. *Inkrementelle und objekto-*

rientierte Vorgehensweisen mit dem V-Modell 97. Oldenbourg, München, 1998.

[Edw97] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proceedings of the IEEE*, volume 85(3), 1997.

[Eke03] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, volume 91(1):pp. 127–144, 2003.

[Eri98] C. Ericsson, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of Nordic Workshop on Programming Theory*, volume 63, 1998.

[Ern03] R. Ernst. Putting It All Together. *ACM Queue*, volume 1(2), 2003.

[Esp08] H. Espinoza, K. Richter, and S. Gérard. Evaluating MARTE in an Industry-Driven Environment: TIMMO's Challenges for AUTOSAR Timing Modeling. *MARTE Workshop at DATE 2008*, March 2008.

[Fei08] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Work. on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*. Barcelona, Espagna, 2008.

[Fen03] W.-C. Feng. Making a Case for Efficient Supercomputing. *Queue*, volume 1(7):pp. 54–64, 2003.

[Fer06] P. Ferrari, A. Flammini, and S. Vitturi. Performance analysis of PROFINET networks. *Computer standards & interfaces*, volume 28(4):pp. 369–385, 2006.

[Fid10] M. Fidler. Survey of deterministic and stochastic service curve models in the network calculus. *Communications Surveys and Tutorials, IEEE*, volume 12(1):pp. 59 – 86, February 2010.

[Fle06] FlexRay Consortium. FlexRay Protocol Specification, Ver. 2.0. `http://www.flexray.de/`, 2006.

[Fly05] M. Flynn and P. Hung. Microprocessor Design Issues: Thoughts on the Road Ahead. *IEEE Micro*, volume 25(3):p. 31, 2005.

[Fre00] Freescale Semiconductors. MPC555 User's Manual and Data Sheet (rev 1). `http://cache..com/files/microcontrollers/doc/user_guide/MPC555UM.pdf?fpsp=1` (retrieved 2010-06-30), October 2000.

[Fre09a] Freescale Semiconductor, Inc. Automotive Powertrain and Body New Product Introduction (NPI). Freescale Technology Forum, `http://www..com.cn/cstory/ftf/2009/download/aut_f0410.pdf`, retrieved 2010-06-23, August 2009.

[Fre09b] Freescale Semiconductors. MPC5668G Single-Chip Automotive Gateway - Fact Sheet. `http://cache..com/files/microcontrollers/doc/fact_sheet/MPC5668GFS.pdf?fpsp=1` (retrieved 2010-06-30), Oktober 2009.

[Gai03] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 189–198, May 2003.

[Gar94] S. Gary, P. Ippolito, G. Gerosa, C. Dietz, J. Eno, and H. Sanchez. PowerPC 603, a microprocessor for portable computers. *IEEE Design & Test of Computers*, volume 11(4):pp. 14–23, 1994.

[Gee05] D. Geer. Chip makers turn to multicore processors. *Computer*, volume 38(5):pp. 11–13, 2005.

[Gei09] M. Geilen. Reduction techniques for synchronous dataflow graphs. In *Proceedings of the 46th Annual Design Automation Conference*, pp. 911–916. ACM, New York, NY, USA, 2009.

[Geo96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. *Rapport de Recherche-Institut National de Recherche en Informatique et en Automatique*, 1996.

[Gho10] A. Ghosal, H. Zeng, M. D. Natale, and Y. Ben-Haim. Computing Robustness of FlexRay Schedules to Uncertainties in Design Parameters. In *Proc. Design Automation and Test in Europe (DATE)*. Dresden, Germany, March 2010.

[Giu01] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pp. 580–589. IEEE Press, Piscataway, NJ, USA, 2001.

[God07] A. Goderis, C. Brooks, I. Altintas, E. Lee, and C. Goble. Composing different models of computation in Kepler and Ptolemy II. *Computational Science–ICCS*, pp. 182–190, 2007.

[Gre93a] K. Gresser. *Echtzeitnachweis Ereignisgesteuerter Realzeitsysteme*. Ph.D. thesis, Technische Universität München, VDI Verlag, 1993.

[Gre93b] K. Gresser. An Event Model for Deadline Verification of Hard Real-Time Systems. In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pp. 118–123. Oulu, Finland, 1993.

[Gri03] K. Grimm. Software technology in an automotive company: major challenges. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pp. 498–503. IEEE Computer Society, Washington, DC, USA, 2003.

[Gua09] N. Guan, M. Stigge, W. Yi, and G. Yu. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. In *Real-Time Systems Symposium (RTSS)*, pp. 387–397. IEEE, 2009.

[Gul07] J. E. Gulliksen and E. Heikkila. Multi-Core Computing in Embedded Applications: Global Market Opportunity and Requirements Analysis. White Paper, Embedded Hardware and Systems Practice Venture Development Corporation, August 2007. `http://www.vdcresearch.com/`.

[Gut97] J. Gutiérrez, J. García, and M. Harbour. On the schedulability analysis for distributed hard real-time systems. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain*, pp. 136–143. Citeseer, 1997.

[Hai07] W. Haid and L. Thiele. Complex Task Activation Schemes in System Level Performance Analysis. In *Proc. of the IEEE/ACM International Conference on HW/SW Codesign and System Synthesis (CODES-ISSS)*, pp. 173–178. Salzburg, Austria, September 2007.

[Ham06] A. Hamann, M. Jersak, K. Richter, and R. Ernst. A Framework for Modular Analysis and Exploration of Heterogeneous Embedded Systems. *Real-Time Systems Journal*, volume 33(1-3):pp. 101–137, July 2006.

[Har87] P. Harter. Response times in level-structured systems. *ACM Trans. Comput. Syst.*, volume 5(3):pp. 232–248, 1987.

[Har01] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Proc. of 13th Euromicro Conference on Real-Time Systems*, pp. 125–134. IEEE Computer Society, 2001.

[Har08] E. Haritan, T. Hattori, H. Yagi, P. Paulin, W. Wolf, A. Nohl, D. Wingard, and M. Muller. Multicore design is the challenge! what is the solution? In *Proceedings of the 45th annual Design Automation Conference*, pp. 128–130. ACM, 2008.

[Hen05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - The Symta/S Approach. In *IEE Proceedings Computers and Digital Techniques*, 2005.

[Hen06a] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *Workshop on Parallel and Distributed Real-Time Systems*. Island of Rhodes, Greece, 2006.

[Hen06b] R. Henia and R. Ernst. Improved offset-analysis using multiple timing-references. *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pp. 450–455, 2006.

[Hen06c] T. Henzinger and S. Matic. An Interface Algebra for Real-Time Components. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, San Jose, CA, April 2006.

[Hen07a] R. Henia, R. Racu, and R. Ernst. Improved Output Jitter Calculation for Compositional Performance Analysis of Distributed Systems. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–8, 2007.

[Hen07b] T. Henriksson, P. van der Wolf, A. Jantsch, and A. Bruce. Network Calculus Applied to Verification of Memory Access Performance in SoCs. In *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA)*. Salzburg, Austria, October 2007.

[Hua07] K. Huang, L. Thiele, T. Stefanov, and E. Deprettere. Performance Analysis of Multimedia Applications using Correlated Streams. In *Design, Automation and Test in Europe (DATE 07)*, pp. 912–917. Nice, France, 2007.

[inf] Infineon Technologies AG.

[Inf08] Infineon Technologies AG. TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1. `http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b73d40837` (retrieved 2010-06-30), January 2008.

[ITR] ITRS. International Technology Roadmap for Semiconductors (2009 edition). `http://public.itrs.net/`.

[Ive07] M. Ivers, B. Janarthanan, and R. Ernst. Predictable Performance on Multithreaded Architectures for Streaming Protocol Processing. In *International Conference on Real-Time and Network Systems (RTNS'07)*, p. 47, 2007.

[Jay07] P. Jayachandran and T. Abdelzaher. A Delay Composition Theorem for Real-Time Pipelines. *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 29–38, 2007.

[Jef93] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pp. 212–221. IEEE, 1993.

[Jer04] M. Jersak, R. Henia, and R. Ernst. Context-Aware Performance Analysis for Efficient Embedded System Design. In *Proc. Design Automation and Test in Europe (DATE)*. Paris, France, mar 2004.

[Jer05] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems*, volume 1(1):pp. 33–49, 2005.

[Jer07] M. Jersak et al. Timing model and methodology for AUTOSAR. *Elektronik Automotive Magazine, Special Issue on AUTOSAR*, 2007.

[Jon08] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic dependencies in modular performance analysis. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT)*, pp. 179–188. ACM, New York, NY, USA, October 2008.

[Jos86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *BCS Computer Journal*, volume 29(5):pp. 390–395, 1986.

[Kai07] Y. Kai and S. Xiaomin. Non-preemptive Scheduling Analysis of Periodic Task Sets with Overheads in an OSEK Compliant System. In *Proceedings of the 2007 International Conference on Convergence Information Technology (ICCIT)*, pp. 1787–1792. IEEE Computer Society, Washington, DC, USA, 2007.

[Kat93] D. Katcher, H. Arakawa, and J. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, pp. 920–934, 1993.

[Kim95] I.-G. Kim, K.-H. Choi, S.-K. Park, D.-Y. Kim, and M.-P. Hong. Real-time scheduling of tasks that contain the external blocking intervals. *RTCSA*, pp. 54–59, 1995.

[Kir08] R. Kirner and P. Puschner. Obstacles in Worst-Case Execution Time Analysis. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pp. 333–339, May 2008.

[Klo10] K. Klobedanz, C. Kuznik, A. Thuy, and W. Mueller. Timing Modeling and Analysis for AUTOSAR-Based Software Development - A Case Study. In *Design Automation and Test in Europe*. Dresden, March 2010.

[Kol06] S. Kollmann, K. Albers, F. Bodmann, and F. Slomka. Modifications on Event Streams for the Real-Time Analysis of Distributed Fixed-Priority Systems. In *ECBS*, pp. 491–492, 2006.

[Kop05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (TTE) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005*, pp. 22–33, 2005.

[Kün06] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proc. Design, Automation and Test in Europe (DATE)*. Munich, Germany, 2006.

[Kün07] S. Künzli, A. Hamann, R. Ernst, and L. Thiele. Combined approach to system level performance analysis of embedded systems. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, p. 68. ACM, 2007.

[Lam09] K. Lampka, S. Perathoner, and L. Thiele. Analytic Real-Time Analysis and Timed Automata: A Hybrid Method for Analyzing Embedded Real-Time Systems. In *EMSOFT '09: Proceedings of the 7th ACM international conference on Embedded software*, pp. 107–116. ACM, New York, NY, USA, 2009.

[LeB01] J. LeBoudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[Lee87] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, volume 75(9), 1987.

[Lee96] C. Lee, J. Hahn, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *17th IEEE Real-Time Systems Symposium, 1996.*, pp. 264–274, 1996.

[Lee98] E. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on computer-aided design of integrated circuits and systems*, volume 17(12):pp. 1217–1229, 1998.

[Lee01] C. Lee, K. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on software engineering*, volume 27(9):pp. 805–826, 2001.

[Lee02] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, volume 35(1):pp. 88–93, 2002.

[Lee03] E. Lee, S. Neuendorffer, and M. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits Systems and Computers*, volume 12(3):pp. 231–260, 2003.

[Lee05] E. Lee. Absolutely positively on time: what would it take?[embedded computing systems]. *Computer*, volume 38(7):pp. 85–87, 2005.

[Leh90] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pp. 201–209, Dec 1990.

[Lei80] H. Leiber, A. Czinczel, and J. Anlauf. Antiblockiersystem (ABS) für Personenkraftwagen. *Bosch Technische Berichte*, volume 7(2), 1980.

[Li95] Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, volume 30(11):pp. 88–98, 1995.

[Li97] Y. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 16(12):p. 1477, 1997.

[Li98] C. Li, R. Bettati, and W. Zhao. Response time analysis for distributed real-time systems with bursty job arrivals. *Proceedings of IEEE ICPP*, 1998.

[Li09] D. Li, X. Sun, J. Wang, and K. Mckinnon. Convergent Lagrangian and domain cut method for nonlinear knapsack problems. *Comput. Optim. Appl.*, volume 42(1):pp. 67–104, 2009.

[Lie97] J. Liedtke, H. Härtig, and M. Hohmuth. OS-Controlled Cache Predictabil-

ity for Real-Time Systems. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, volume 0:p. 213, 1997.

[Liu73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, volume 20(1):pp. 46–61, 1973.

[Liu08] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 2008.

[LT02] M. G. L. Thiele, S. Chakraborty and S. Künzli. Design space exploration of network processor architectures. *Network Processor Design: Issues and Practices*, volume 1, October 2002. Morgan Kaufmann Publishers.

[Mac98] I. MacKenzie. *The 8051 Microcontroller*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1998.

[Man07] S. Manolache, P. Eles, and Z. Peng. Fault-aware communication mapping for NoCs with guaranteed latency. *International Journal of Parallel Programming*, volume 35(2):pp. 125–156, 2007.

[Mar90] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley and Sons Ltd., 1990.

[Mar08] A. Martin. New debugging concept for symmetric multiprocessing (SMP). White paper Lauterbach GmbH, `http://www.lauterbach.com/`, Feb 2008.

[Mat97] D. Matzke et al. Will physical scalability sabotage performance gains? *Computer*, volume 30(9):pp. 37–39, 1997.

[Max04] A. Maxiaguine, S. Künzli, and L. Thiele. Workload Characterization Model for Tasks with Variable Execution Demand. In *Proc. Design Automation and Test in Europe (DATE)*. Paris, France,, 2004.

[May08] S. Mayer and R. Pleines. Mega Market for Ulta-Low-Cost Cars - Focusing on customers in developing markets. A.T. Kearney, Stuttgart, 2008.

[Moh08] S. Mohalik, A. C. Rajeev, M. G. Dixit, S. Ramesh, P. V. Suman, P. K. Pandya, and S. Jiang. Model checking based analysis of end-to-end latency in embedded, real-time systems with clock drifts. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pp. 296–299. ACM, New York, NY, USA, 2008.

[Mok97] A. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, volume 23(10):pp. 635–645, 1997.

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, volume 38:p. 114–117, 1965.

[Moo05] A. Moonen, R. Van Den Berg, M. Bekooij, H. Bhullar, and J. Van Meerbergen. A multi-core architecture for in-car digital entertainment. In *Proc. of GSPx Conference*. Citeseer, 2005.

[Mor07] O. Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, volume 2007:p. 14, 2007.

[Mün00] A. Münnich and G. Färber. Calculating Worst-Case Execution Times of Transactions in Databases for Event-Driven, Hard Real-Time Embedded Systems. In *IDEAS*, pp. 149–157, 2000.

[Neg03] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 201–206. ACM, New York, NY, USA, 2003.

[Neg09] M. Negrean, S. Schliecker, and R. Ernst. Response-Time Analysis of Arbitrarily Activated Tasks in Multiprocessor Systems with Shared Resources. In *In Proc. of Design, Automation, and Test in Europe (DATE)*. Nice, France, April 2009.

[Neg10] M. Negrean, S. Schliecker, and R. Ernst. Timing Implications of Sharing Resources in Multicore Real-Time Automotive Systems. In *SAE World Congress*. SAE International, Detroit, MI, USA, April 2010.

[Nor99] C. Norström, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, volume 182. IEEE Computer Society Washington, DC, USA, 1999.

[Not10] J. Noto, G. Fenical, and C. Tong. Automotive EMI shielding: Controlling automotive electronic emissions and susceptibility with proper EMI suppression methods. *Automotive DesignLine*, April 2010. `http://www.automotivedesignline.com/224400504` (retrieved 2010-05-10).

[Obe09] R. Obermaisser, C. E. Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 28(7):pp. 956–965, 2009.

[OSE05] OSEK/VDX Consortium. OSEK/VDX Operating System. V.2.2.3. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, retrieved 2010-06-23, February 2005.

[Pag07] M. Paganini. Nomadik®: A Mobile Multimedia Application Processor Platform. *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, pp. 749–750, 2007.

[Pal98] J. Palencia and M. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS98)*, 1998.

[Pau02] P. Paulin, C. Pilkington, and E. Bensoudane. StepNP: a system-level ex-

ploration platform for network processors. *Design & Test of Computers, IEEE*, volume 19(6):pp. 17–26, 2002.

[Pau04] P. Paulin, C. Pilkington, E. Bensoudane, M. Langevin, and D. Lyonnard. Application of a multi-processor SoC platform to high-speed packet forwarding. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, volume 3, 2004.

[Pau06] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, D. Lavigueur, B.; Lo, G. Beltrame, V. Gagne, and G. Nicolescu. Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2006.

[Pau09] P. Paulin, O. Benny, M. Langevin, Y. Bouchebaba, C. Pilkington, B. Lavigueur, D. Lo, V. Gagne, and M. Metzger. MPSoC Platform Mapping Tools for Data-Dominated Applications. *Model-Based Design for Embedded Systems*, p. 179, 2009.

[Pel10a] R. Pellizzoni and M. Caccamo. Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems. *IEEE Transactions on Computers*, volume 59(3):pp. 400–415, March 2010.

[Pel10b] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *ACM/IEEE Conference of Design, Automation, and Test in Europe (DATE)*, 2010.

[Per08] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of Different Abstractions on the Performance Analysis of Distributed Hard Real-Time Systems. *Design Automation for Embedded Systems*, pp. 1–23, April 2008.

[Pet01] S. Petters. and G. Färber. Scheduling analysis with respect to hardware related preemption delay. In *In Workshop on Real-Time Embedded Systems, London, United Kingdom, December*, volume 3, 2001.

[Pet06] P. Peti, R. Obermaisser, and H. Paulitsch. Investigating Connector Faults in the Time-Triggered Architecture. In *Proc. IEEE Conf. on Emerging Technologies and Factory Automation (ETFA)*, pp. 887–896. Citeseer, 2006.

[Pha07] L. T. X. Phan, S. Chakraborty, P. S. Thiagarajan, and L. Thiele. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In *Proc. of the 28th IEEE Real-Time Systems Symposium (RTSS)*, p. 343–352. IEEE Computer Society, Tucson, AZ, US, dec 2007.

[Pop03] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. *Design*

Automation and Test in Europe Conference and Exhibition (DATE), pp. 184–189, 2003.

[Pop06] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing Analysis of the FlexRay Communication Protocol. In *Proceedings of 18th EuroMicro Conference on Real-Time Systems, Dresden*, pp. 203–213, 2006.

[Pop09] K. Popovici and A. Jerraya. Programming Models for MPSoC. *Model-Based Design for Embedded Systems*, p. 231, 2009.

[Pua02] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pp. 114–123. Citeseer, 2002.

[Pus08] P. Puschner and M. Schoeberl. On composable system timing, task timing, and WCET analysis. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Prague, Czech Republic, June 2008.

[Rac07] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst. Improved response time analysis of tasks scheduled under preemptive Round-Robin. In *Proc. of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES-ISSS)*, pp. 179–184. ACM Press New York, NY, USA, Salzburg, Austria, 2007.

[Rac08] R. Racu. *Performance Characterization and Sensitvity Analysis of Real-Time Embedded Systems*. Ph.D. thesis, Technische Universität Braunschweig, 2008.

[Raj88] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. *Real-Time Systems Symposium, 1988., Proceedings.*, pp. 259–269, 1988. Was: RSL88.

[Raj91] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[Raz94] R. Razdan and M. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pp. 172–180. ACM, 1994.

[Ric02a] K. Richter and R. Ernst. Event Model Interfaces for Heterogeneous System Analysis. In *Proc. Design Automation and Test in Europe (DATE)*. Paris, France, March 2002.

[Ric02b] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model Composition for Scheduling Analysis in Platform Design. In *Proceedings 39th Design Automation Conference (DAC 2002)*, June 2002.

[Ric03] K. Richter, R. Racu, and R. Ernst. Scheduling Analysis Integration for

Heterogeneous Multiprocessor SoC. In *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS)*. Cancun, Mexico, December 2003.

[Ric04] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. Ph.D. thesis, Technische Universität Braunschweig, 2004.

[Ric06] K. Richter, M. Jersak, and R. Ernst. How OEMs and Suppliers can tackle the Network Integration Challenges. In *In Proc. Embedded Real-Time Software Congress (ERTS)*. Toulouse, France, January 2006.

[Rod10] S. Rodt, B. Georgi, B. Huckestein, L. Mönch, R. Herbener, H. Jahn, K. Koppe, and J. Lindmaier. CO2-Emissionsminderung im Verkehr in Deutschland - Mögliche Maßnahmen und ihre Minderungspotenziale. `http://www.umweltbundesamt.de`, Umweltbundesamtes, Dessau-Roßlau, mar 2010.

[Ros07] J. Rosen, A. Andrei, P. Eles, and Z. Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. Real-Time Systems Symposium (RTSS)*, pp. 49–60, 2007.

[Rox08] J. Rox and R. Ernst. Modeling event stream hierarchies with hierarchical event models. In *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pp. 492–497. ACM New York, NY, USA, 2008.

[Rox10] J. Rox and R. Ernst. Exploiting Inter-Event Stream Correlations Between Output Event Streams of non-Preemptively Scheduled Tasks. In *Proc. Design, Automation and Test in Europe (DATE 2010)*, March 2010.

[San07] O. Sander, J. Becker, M. Hubner, M. Dreschmann, J. Luka, M. Traub, and T. Weber. Modular system concept for a FPGA-based Automotive Gateway. *VDI BERICHTE*, volume 2000:p. 221, 2007.

[Sch97] R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, volume 34(6):pp. 52–59, 1997.

[Sch05] H. Schiøler, J. J. Jessen, J. D. Nielsen, and K. G. Larsen. Network Calculus for Real Time Analysis of Embedded Systems with Cyclic Task Dependencies. In *Computers and Their Applications*, pp. 326–332, 2005.

[Sch06a] S. Schliecker, M. Ivers, and R. Ernst. Integrated analysis of communicating tasks in MPSoCs. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (Codes-ISSS)*, pp. 288–293. ACM Press New York, NY, USA, Seoul, Korea, 2006.

[Sch06b] S. Schliecker, M. Ivers, and R. Ernst. Memory Access Patterns for the Analysis of MPSoCs. In *IEEE North-East Workshop on Circuits and Systems (NewCAS)*, pp. 249–252, 2006.

[Sch06c] S. Schliecker, M. Ivers, J. Staschulat, and R. Ernst. A Framework for the

Busy Time Calculation of Multiple Correlated Events. In *6th Intl. Worst Case Execution Time Workshop*. Dresden, Germany, 2006.

[Sch07] S. Schliecker, S. Stein, and R. Ernst. Performance Analysis of Complex Systems by Integration of Dataflow Graphs and Compositional Performance Analysis. In *Proc. Design, Automation and Test in Europe (DATE)*. Nice, France, April 2007.

[Sch08a] G. Schirner and R. Dömer. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *ACM Transactions on Embedded Computing Systems (TECS)*, volume 8(1):p. 4, 2008.

[Sch08b] S. Schliecker, M. Negrean, G. Nicolescu, P. Paulin, and R. Ernst. Reliable Performance Analysis of a Multicore Multithreaded System-On-Chip. In *Proc. 6th International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS)*. Atlanta, GA, October 2008.

[Sch08c] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In *Proc. 6th International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS)*. Atlanta, GA, October 2008.

[Sch08d] S. Schliecker, S. Stein, J.-C. Braam, and M. Schnieringer. System Level Performance Analysis with Formal Methods and Virtual Prototyping, Embedded World Conference, Nürnberg, Germany. In *Embedded World Conference*, February 2008.

[Sch08e] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Proceedings of the 45th annual Design Automation Conference*, pp. 290–295. ACM, 2008.

[Sch09a] B. Schaden and U. Schäfer. Trendanalyse Electronic Components and Systems 2009/10 until 2013. Technical report, ZVEI - Zentralverband Elektrotechnik und Elektronikindustrie e.V., Frankfurt am Main, dec 2009. `http://www.zvei.org/`.

[Sch09b] S. Schliecker, M. Negrean, and R. Ernst. Response Time Analysis in Multicore ECUs with Shared Resources. *IEEE Transactions on Industrial Informatics*, volume 5(4), November 2009.

[Sch09c] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System Level Performance Analysis for Real-Time Automotive Multi-Core and Network Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 28(7):pp. 979–992, July 2009.

[Sch09d] M. Schoeberl and P. Puschner. Is Chip-Multiprocessing the End of Real-Time Scheduling? In *Proc. 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*. OCG, Dublin, Ireland, July 2009.

[Sch10a] S. Schliecker and R. Ernst. Real-Time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Transactions on Embedded*

Computing Systems (Special Issue on Model Driven Embedded System Design), 2010. (to appear).

[Sch10b] S. Schliecker, M. Negrean, and R. Ernst. Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems. In *Proc. Design, Automation and Test in Europe (DATE)*. Dresden, Germany, March 2010.

[Sch10c] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pp. 332–337. ACM, New York, NY, USA, 2010.

[Seb09] M. Sebastian and R. Ernst. Reliability and Safety Guarantees in Modern MPSoCs with Real- Time Requirements. In *3rd edaWorkshop*. Dresden, Germany, May 2009.

[Seg98] S. Segars. The ARM9 family-high performance microprocessors for embedded applications. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pp. 230–235, 1998.

[Sha90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, volume 39(9):pp. 1175–1185, Sep 1990.

[Shi02] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pp. 389–398, 2002.

[Smo06] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 223–234. IEEE Computer Society, Washington, DC, USA, 2006.

[Sri00] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 2000.

[Sta05a] J. Staschulat. SYMTA/P - Performance Verification for Complex Embedded Systems v1.2. `http://sourceforge.net/projects/symtap/`, August 2005.

[Sta05b] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 41–48, 2005.

[Sta05c] J. Staschulat, S. Schliecker, M. Ivers, and R. Ernst. Analysis of Memory Latencies in Multi-Processor Systems. In *5th Intl. Worst Case Execution Time Workshop*, 2005.

[Sta07] J. Staschulat and R. Ernst. Scalable Precision Cache Analysis for Real-time Software. *ACM Transactions on Embedded Computing Systems*, volume 6(4), Sep. 2007. Special Issue LCTES 05.

[Sta09a] J. Staschulat and M. Bekooij. Dataflow models for shared memory access latency analysis. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pp. 275–284. ACM, New York, NY, USA, 2009.

[Sta09b] Statistisches Bundesamt. Umweltnutzung und Wirtschaft - Bericht zu den Umweltökonomischen Gesamtrechnungen. `http://www.destatis.de` (retrieved 5-5-2010), Wiesbaden, November 2009.

[Ste06] S. Stein, A. Hamann, and R. Ernst. Real-time property verification in organic computing systems. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Cyprus, 2006.

[Ste08] S. Stein, J. Diemer, M. Ivers, S. Schliecker, and R. Ernst. On the Convergence of the SymTA/S analysis. Technical report, Technische Universität Braunschweig, Braunschweig, Germany, November 2008.

[Ste09] M. Steine, M. Bekooij, and M. Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pp. 37–44. IEEE Computer Society, 2009.

[Ste10] S. Stein, M. Ivers, J. Diemer, and R. Ernst. A polynomial time algorithm for computing response time bounds in static priority scheduling with convex event models. In *Euromicro Conference on Real-Time Systems (ECRTŚ10)*, July 2010.

[Ste11] S. Stein, S. Schliecker, J. Diemer, M. Ivers, and R. Ernst. A CPO Semantics and Proof of Convergence for Compositional Performance Analysis. *Springer, Theoretical Computer Science*, (under review) 2011. See also [Ste08] for partial preprint.

[Sti98] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, volume 6(5):p. 624, 1998.

[Sto05] J. Stohr, A. von Bulow, and G. Färber. Bounding worst-case access times in modern multiprocessor systems. In *Proc. 17th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 189 – 198, july 2005.

[Sun95] J. Sun and J. Liu. Bounding the end-to-end response time in multiprocessor real-time systems. *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 1995.

[SV07] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, volume 95(3):pp. 467–506, 2007.

[Swi00] J. Swingler, J. McBride, and C. Maul. Degradation of road tested automotive connectors. *IEEE Transactions on Components and Packaging Technologies*, volume 23(1):pp. 157–164, 2000.

[TAD09] TIMMO Project Deliverable D6: TADL: Timing Augmented Description Language version 2. `http://www.timmo.org/` (retrieved 2010-0508), 2009.

[Tar55] A. Tarski. A Lattice-Theoretical fixpoint theorem. *Pacific J. Math.*, volume 5:pp. 285–309, 1955.

[Tex04] Texas Instruments. OMAP 1 Processors: OMAP1710. `http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=11991&contentId=4670`, retrieved 2010-08-27, 2004.

[Thi00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pp. 101–104 vol.4. Geneva, Switzerland, 2000.

[Thi02] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. *Network Processor Design: Issues and Practices*, volume 1:pp. 55–89, 2002.

[Thi06] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pp. 34–43. ACM, New York, NY, USA, 2006.

[Thi09] L. Thiele and N. Stoimenov. Modular Performance Analysis of Cyclic Dataflow Graphs. In *EMSOFT 09: Proceedings of the 7th ACM international conference on Embedded software*, pp. 127–136. Grenoble, France, 2009.

[Tin94a] K. Tindell. Adding time-offsets to schedulability analysis. Technical report, University of York, 1994.

[Tin94b] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, volume 6(2):pp. 133–151, 1994.

[Tin94c] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, volume 40(2-3):pp. 117–134, 1994.

[Val97] C. Valderrama, A. Changuel, and A. Jerraya. Virtual prototyping for modular and flexible hardware-software systems. *Design Automation for Embedded Systems*, volume 2(3):pp. 267–282, 1997.

[Wan05] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characteriza-

tion of event streams in analysis of hard real-time applications. *Real-Time Systems*, volume 29(2):pp. 205–225, 2005.

[Wan06a] E. Wandeler. *Modular Performance Analysis and Interface-based Design of Embedded Systems*. Ph.D. thesis, Swiss Federal Institute of Technology (ETH), 2006.

[Wan06b] E. Wandeler, A. Maxiaguine, and L. Thiele. Performance analysis of greedy shapers in real-time systems. *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pp. 444–449, 2006.

[Wan06c] E. Wandeler and L. Thiele. Interface-Based Design of Real-Time Systems with Hierarchical Scheduling. In *Real-Time and Embedded Technology and Applications Symposium, IEEE*, volume 0, pp. 243–252. IEEE Computer Society, Los Alamitos, CA, USA, 2006.

[Wei07] X. Wei, L. Guzzella, V. Utkin, and G. Rizzoni. Model-based fuel optimal control of hybrid electric vehicle using variable structure control systems. *Journal of Dynamic Systems, Measurement, and Control*, volume 129:p. 13, 2007.

[Wel09] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-mode multicore reliability. *SIGPLAN Not.*, volume 44(3):pp. 169–180, 2009.

[Wig07] M. Wiggers, M. Bekooij, and G. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, p. 22. ACM, 2007.

[Wig09] M. H. Wiggers, M. J. Bekooij, and G. J. Smit. Monotonicity and run-time scheduling. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pp. 177–186. ACM, New York, NY, USA, 2009.

[Wil08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, volume 7(36):pp. 1–53, April 2008.

[Wil09] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 28(7):pp. 966–978, July 2009.

[Wit51] E. Witt. Beweisstudien zum Satz von M. Zorn. *Mathematische Nachrichten*, volume 4:p. 434–438, 1951.

[Won10] S. Wonneberger, T. Graf, H. Sahlbach, S. Whitty, O. Bende, and R. Ernst.

Real-time Image Processing for Camera-based Driver Assistance Applications. In *Automation, Assistence and Embedded Real Time Platforms for Transportation (AAET)*, February 2010.

[Wul95] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Archiecture News*, volume 23(1):pp. 20–24, 1995.

[ZVE09] ZVEI - Zentralverband Elektrotechnik- und Elektronikindustrie e. V. Embedded Software & Systems in der Elektrotechnik- und Elektronikindustrie. `http://www.zvei.org` (retrieved 2010-04-26), Frankfurt am Main, Germany, November 2009.